Modularity Graph Clustering on the GPU

B. O. Fagginger Auer R. H. Bisseling

Utrecht University

April 10, 2012

Introduction

- We will perform greedy clustering of large graphs.

Introduction

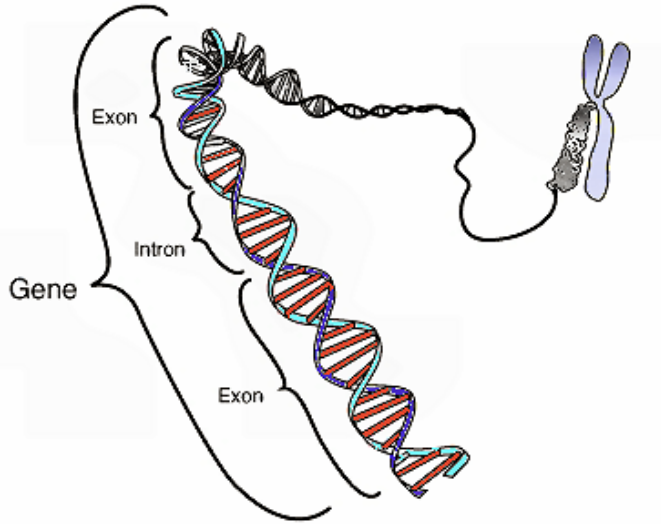
- We will perform greedy clustering of large graphs.
- Clustering $\approx$ isolating 'related' groups of vertices in a graph.

Introduction

- We will perform greedy clustering of large graphs.
- Clustering $\approx$ isolating 'related' groups of vertices in a graph.
- Our primary interests are speed and parallelisation.

Applications of Clustering

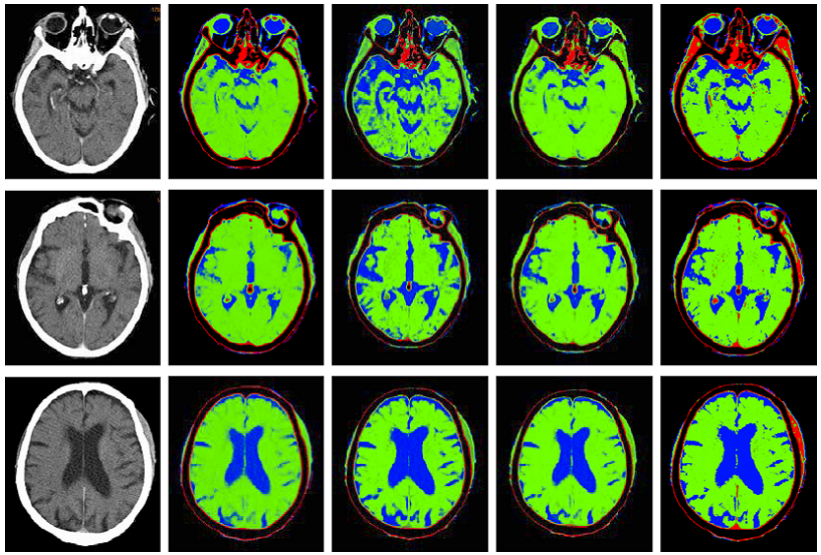
- Bioinformatics: group genes with similar expressions.



© National Human Genome Research Institute

Applications of Clustering

- Bioinformatics: group genes with similar expressions.
- Medical imaging: image segmentation in CT/MRI.



© Alexandra Lauric

Applications of Clustering

- Bioinformatics: group genes with similar expressions.
- Medical imaging: image segmentation in CT/MRI.
- Market research: consumer grouping.

© Georgia Tech

Applications of Clustering

- Bioinformatics: group genes with similar expressions.
- Medical imaging: image segmentation in CT/MRI.
- Market research: consumer grouping.
- Social networks: community detection.

Social Media Landscape

© Fred Cavazza

Modularity

- Let $G = (V, E)$ be a graph with vertices V, edges E, and edge weights $\omega : E \rightarrow \mathbb{R}_{>0}$.

Modularity

- Let $G = (V, E)$ be a graph with vertices V, edges E, and edge weights $\omega : E \to \mathbb{R}_{>0}$.
- The weight $\omega(\{u, v\})$ measures the strength of the link between vertices u and v.

Modularity

- Let $G = (V, E)$ be a graph with vertices V, edges E, and edge weights $\omega : E \to \mathbb{R}_{>0}$.
- The weight $\omega(\{u, v\})$ measures the strength of the link between vertices u and v.
- A clustering of G is a partitioning $\mathcal{C}$ of V:

$$V = \bigcup_{C \in \mathcal{C}} C \qquad \text{as a disjoint union.}$$

Modularity

- Let $G = (V, E)$ be a graph with vertices V, edges E, and edge weights $\omega : E \to \mathbb{R}_{>0}$.
- The weight $\omega(\{u, v\})$ measures the strength of the link between vertices u and v.
- A clustering of G is a partitioning $\mathcal{C}$ of V:

$$V = \bigcup_{C \in \mathcal{C}} C \qquad \text{as a disjoint union.}$$

- Quality of a clustering is measured by its modularity $\mathrm{mod}(\mathcal{C})$, introduced in 2004 by Newman and Girvan.

Modularity clustering

- Clustering modularity is defined as

$$\mathrm{mod}(\mathcal{C}) := \frac{\sum_{C \in \mathcal{C}} \sum_{\substack{\{u,v\} \in E \\ u,v \in C}} \omega(\{u,v\})}{\sum_{e \in E} \omega(e)} - \frac{\sum_{C \in \mathcal{C}} \left(\sum_{v \in C} \zeta(v) \right)^2}{4 \left(\sum_{e \in E} \omega(e) \right)^2}.$$

Modularity clustering

- Clustering modularity is defined as

$$\mathrm{mod}(\mathcal{C}) := \frac{\sum_{C \in \mathcal{C}} \sum_{\substack{\{u,v\} \in E \\ u,v \in C}} \omega(\{u,v\})}{\sum_{e \in E} \omega(e)} - \frac{\sum_{C \in \mathcal{C}} \left(\sum_{v \in C} \zeta(v) \right)^2}{4 \left(\sum_{e \in E} \omega(e) \right)^2}.$$

- Here, vertex weights $\zeta : V \to \mathbb{R}_{\geq 0}$ are defined as

$$\zeta(v) := \sum_{\{u,v\} \in E} \omega(\{u,v\}).$$

Modularity clustering

- Clustering modularity is defined as

$$\mathrm{mod}(\mathcal{C}) := \frac{\sum_{C \in \mathcal{C}} \sum_{\substack{\{u,v\} \in E \\ u,v \in C}} \omega(\{u,v\})}{\sum_{e \in E} \omega(e)} - \frac{\sum_{C \in \mathcal{C}} \left(\sum_{v \in C} \zeta(v) \right)^2}{4 \left(\sum_{e \in E} \omega(e) \right)^2}.$$

- Here, vertex weights $\zeta : V \to \mathbb{R}_{\geq 0}$ are defined as

$$\zeta(v) := \sum_{\{u,v\} \in E} \omega(\{u,v\}).$$

- We have $-\frac{1}{2} \leq \mathrm{mod}(\mathcal{C}) \leq 1$.

Modularity clustering

- $\mathrm{mod}(\mathcal{C})$ can be rewritten to

$$\frac{1}{4\,\Omega^2} \sum_{C \in \mathcal{C}} \left[\zeta(C)\,(2\,\Omega - \zeta(C)) - 2\,\Omega \left(\sum_{\substack{C' \in \mathcal{C} \\ C' \neq C}} \omega(\mathrm{cut}(C, C')) \right) \right].$$

Modularity clustering

- $\mathrm{mod}(\mathcal{C})$ can be rewritten to

$$\frac{1}{4\,\Omega^2} \sum_{C \in \mathcal{C}} \left[\zeta(C)\,(2\,\Omega - \zeta(C)) - 2\,\Omega \left(\sum_{\substack{C' \in \mathcal{C} \\ C' \neq C}} \omega(\mathrm{cut}(C, C')) \right) \right].$$

- Here,

$$\begin{aligned} \Omega &:= \sum_{e \in E} \omega(e), \\ \zeta(v) &:= \sum_{\{u,v\} \in E} \omega(\{u, v\}), \\ \mathrm{cut}(C, C') &:= \{\{u, v\} \in E \mid u \in C \text{ and } v \in C'\}. \end{aligned}$$

Modularity clustering

- $\mathrm{mod}(\mathcal{C})$ can be rewritten to

$$\frac{1}{4\,\Omega^2} \sum_{C \in \mathcal{C}} \left[\zeta(C)\,(2\,\Omega - \zeta(C)) - 2\,\Omega \left(\sum_{\substack{C' \in \mathcal{C} \\ C' \neq C}} \omega(\mathrm{cut}(C, C')) \right) \right].$$

- Here,

$$\begin{aligned} \Omega &:= \sum_{e \in E} \omega(e), \\ \zeta(v) &:= \sum_{\{u,v\} \in E} \omega(\{u, v\}), \\ \mathrm{cut}(C, C') &:= \{\{u, v\} \in E \mid u \in C \text{ and } v \in C'\}. \end{aligned}$$

- To calculate modularity, we only need to keep track of summed vertex weights of clusters and summed edge weights between clusters.

Merging clusters

- Merge two clusters $C, C' \in \mathcal{C}$ to a single larger cluster $C \cup C'$.

Merging clusters

- Merge two clusters $C, C' \in \mathcal{C}$ to a single larger cluster $C \cup C'$.
- Then,

$$\zeta(C \cup C') = \zeta(C) + \zeta(C')$$

Merging clusters

- Merge two clusters $C, C' \in \mathcal{C}$ to a single larger cluster $C \cup C'$.
- Then,

$$\zeta(C \cup C') = \zeta(C) + \zeta(C')$$

and the modularity is increased by (eq. (4) of Ovelgönne et al., 2010)

$$\frac{1}{2\,\Omega^2}\left(2\,\Omega\,\omega(\mathrm{cut}(C, C')) - \zeta(C)\,\zeta(C')\right).$$

Merging clusters

- Merge two clusters $C, C' \in \mathcal{C}$ to a single larger cluster $C \cup C'$.
- Then,

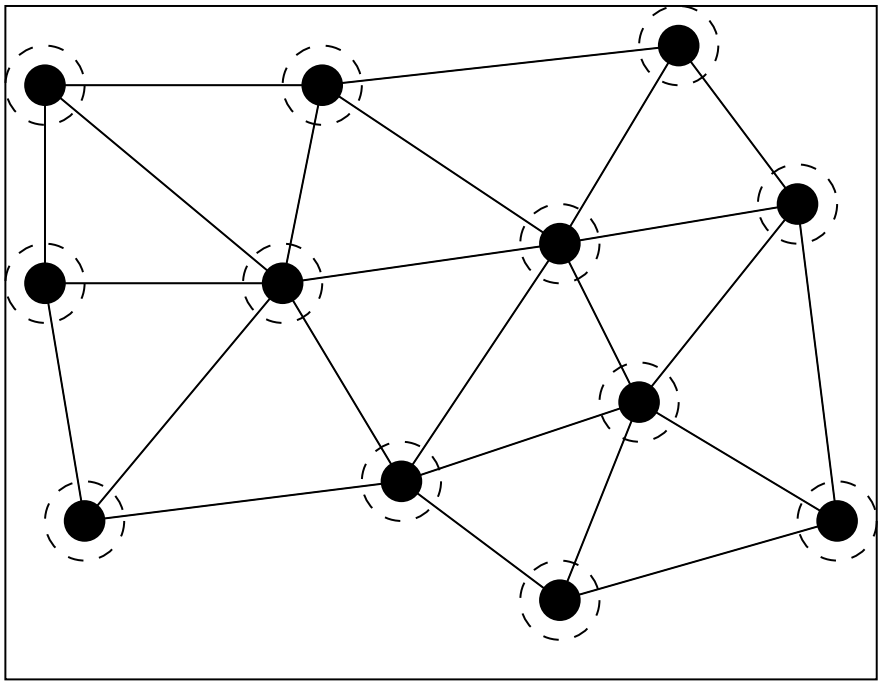
$$\zeta(C \cup C') = \zeta(C) + \zeta(C')$$

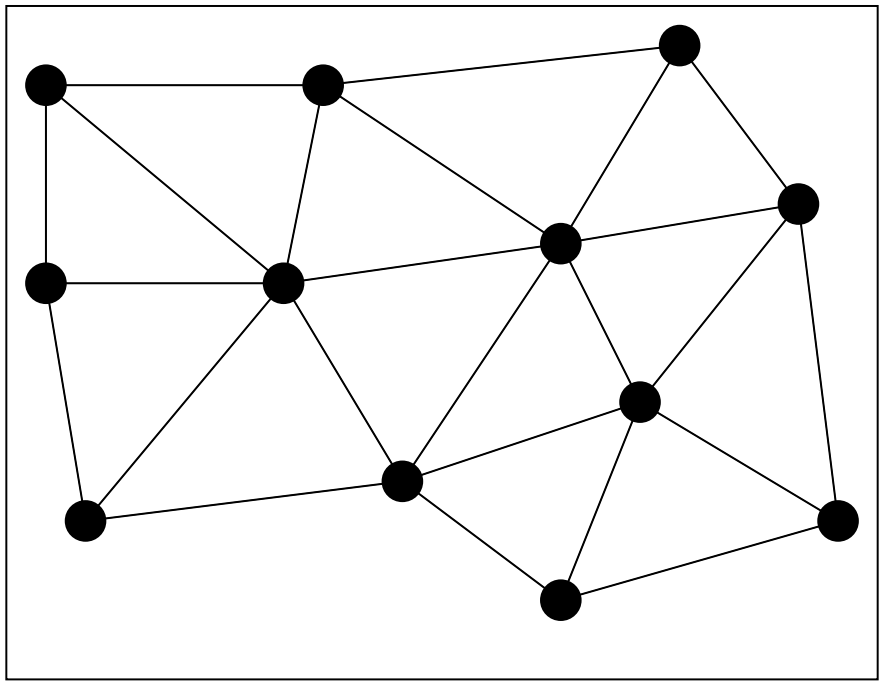
and the modularity is increased by (eq. (4) of Ovelgönne et al., 2010)

$$\frac{1}{2\,\Omega^2}\left(2\,\Omega\,\omega(\mathrm{cut}(C, C')) - \zeta(C)\,\zeta(C')\right).$$

- This suggests a greedy agglomerative strategy (e.g. Zhu et al., 2008).

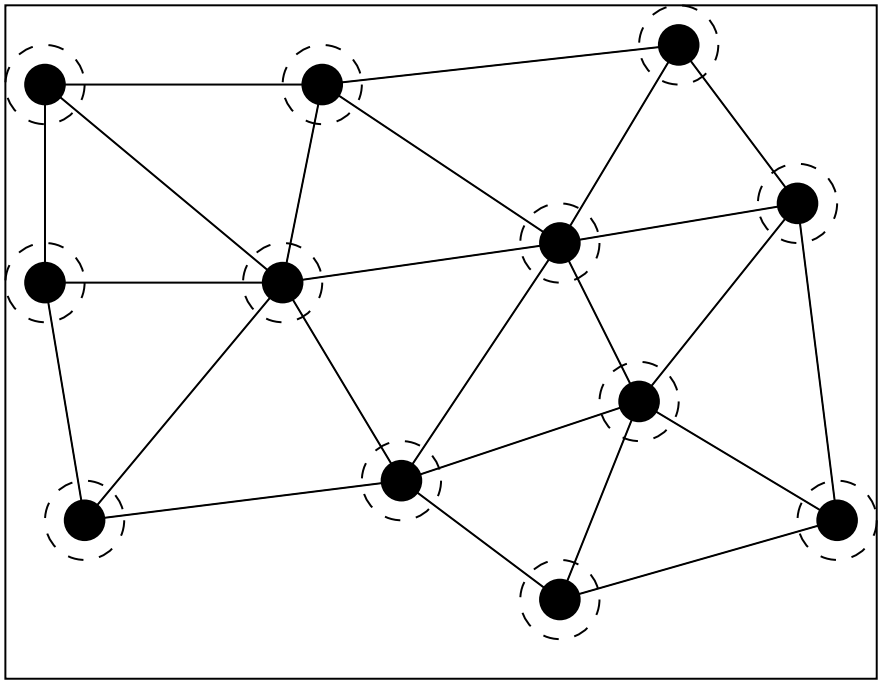
Agglomerative clustering

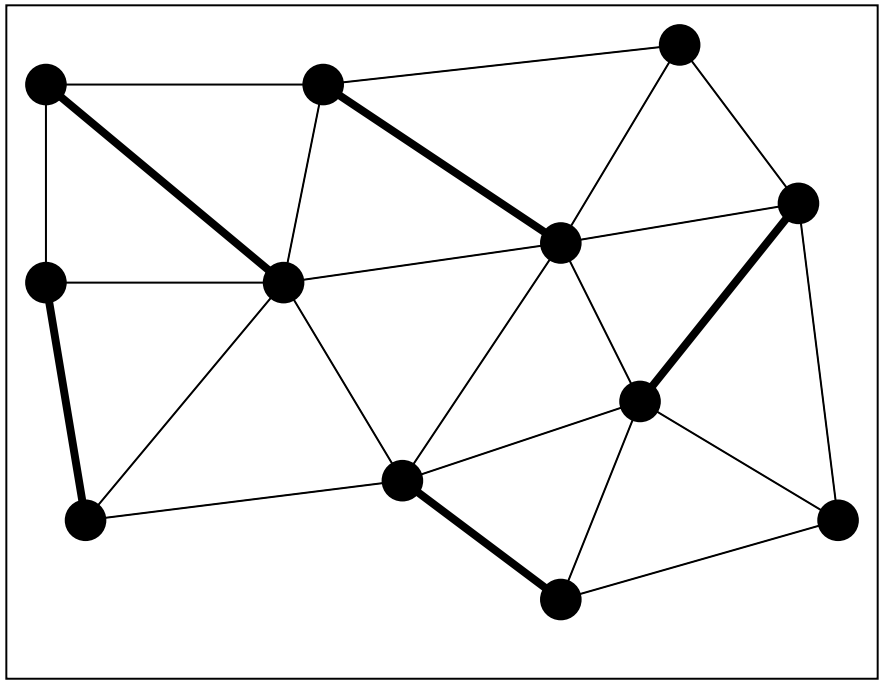




Start with vertices in a separate cluster: $\mathcal{C} = \{\{v\} \mid v \in V\}$.

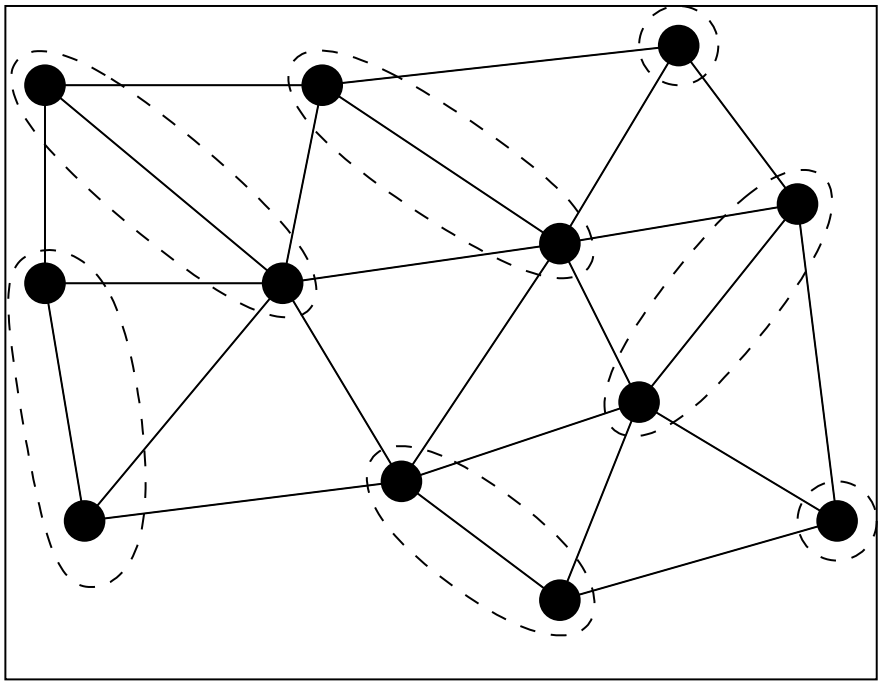
Agglomerative clustering

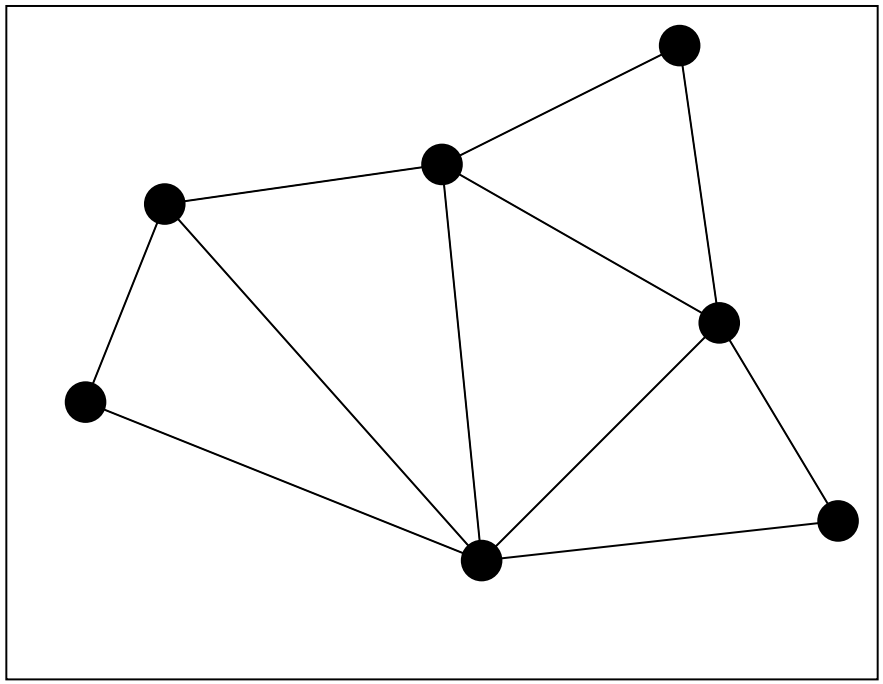




Match clusters to increase modularity.

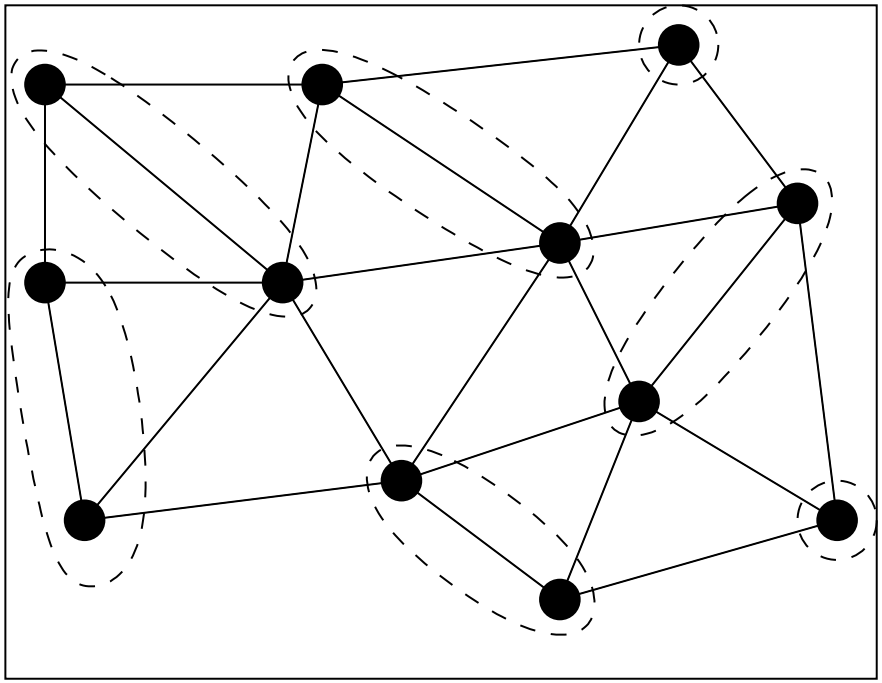
Agglomerative clustering

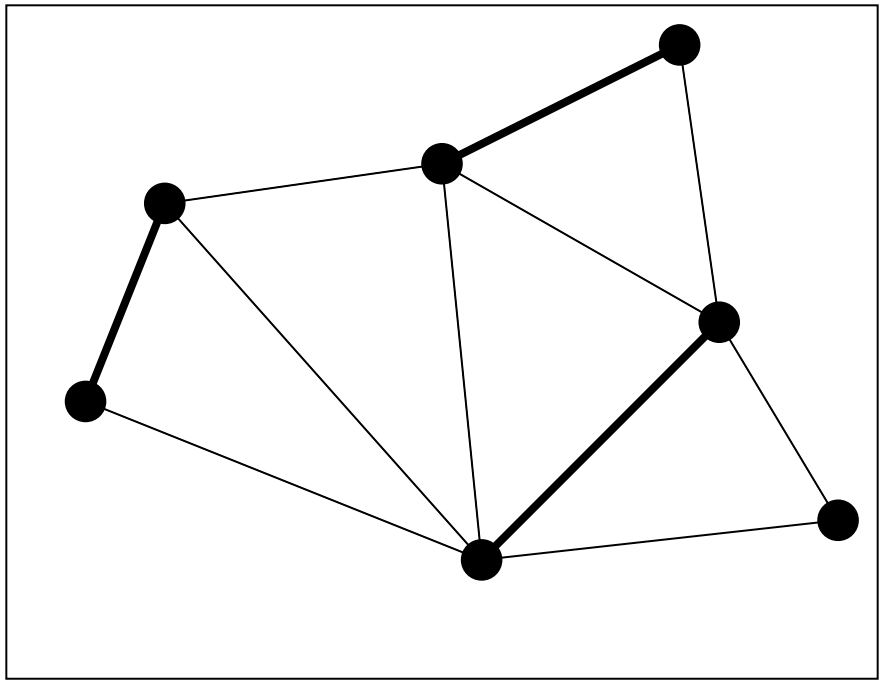




Merge matched clusters (sum ω and ζ).

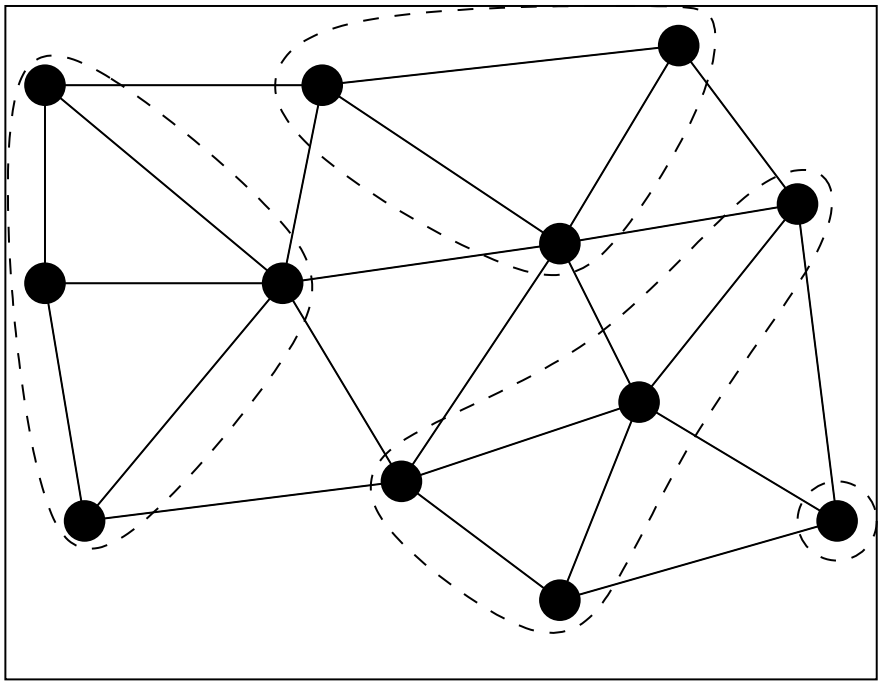
Agglomerative clustering

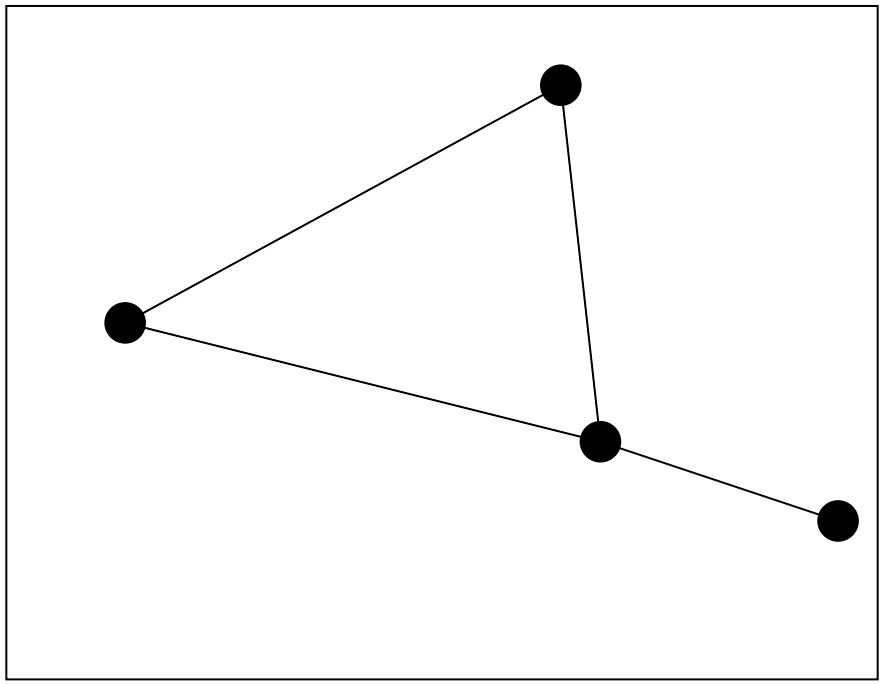




Match clusters to increase modularity.

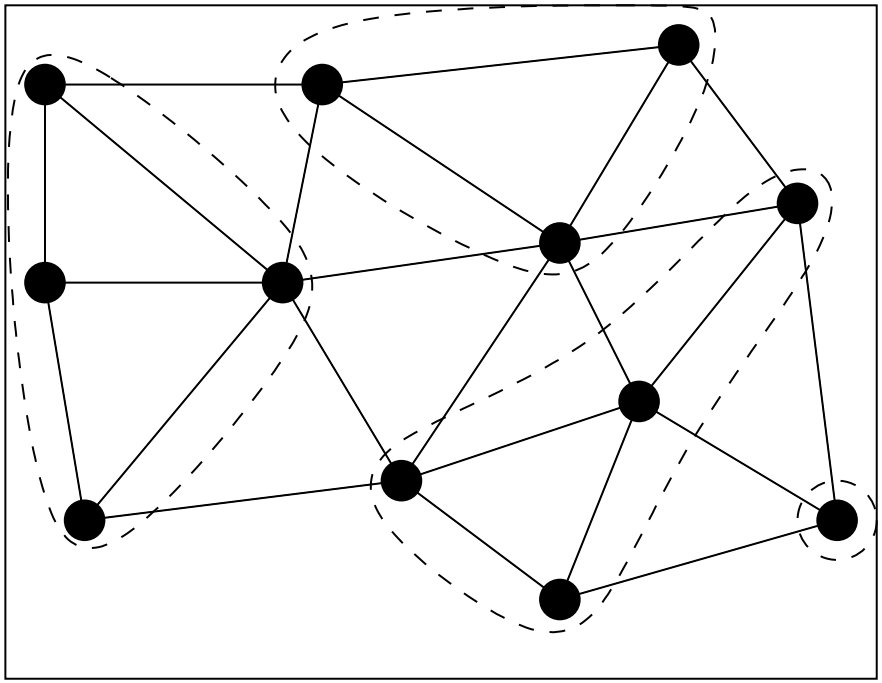
Agglomerative clustering

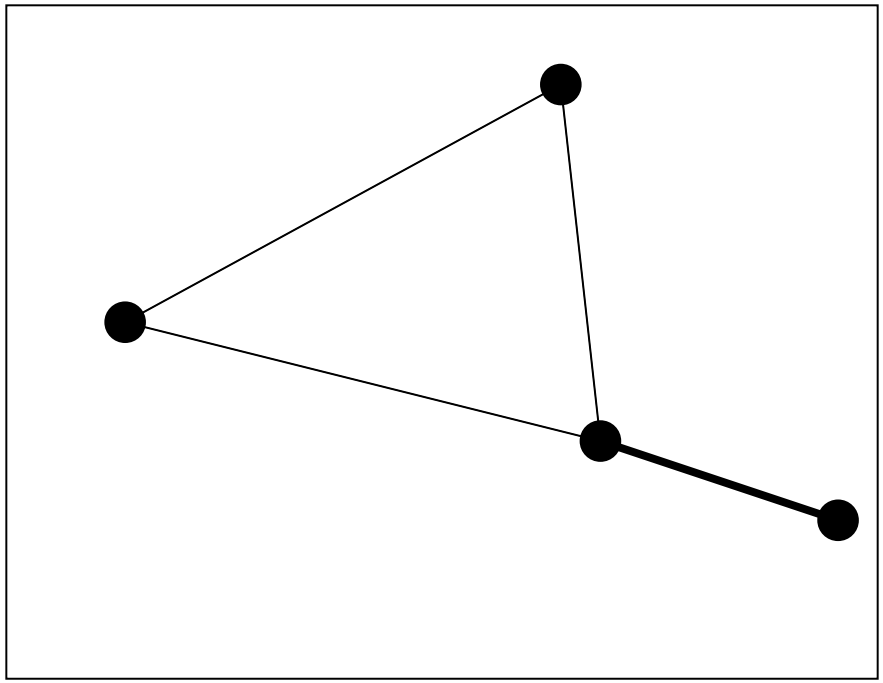




Merge matched clusters (sum ω and ζ).

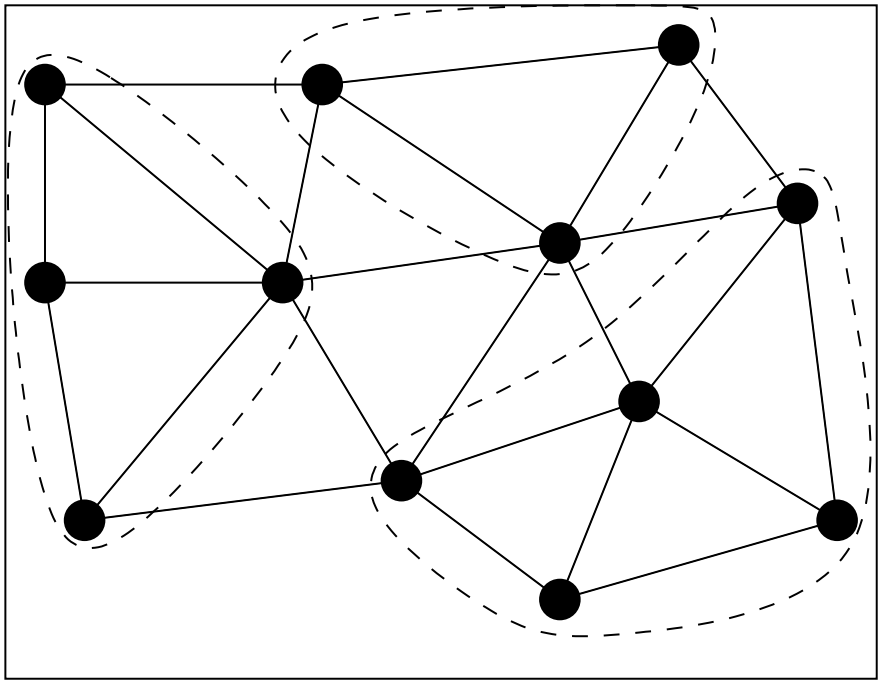
Agglomerative clustering

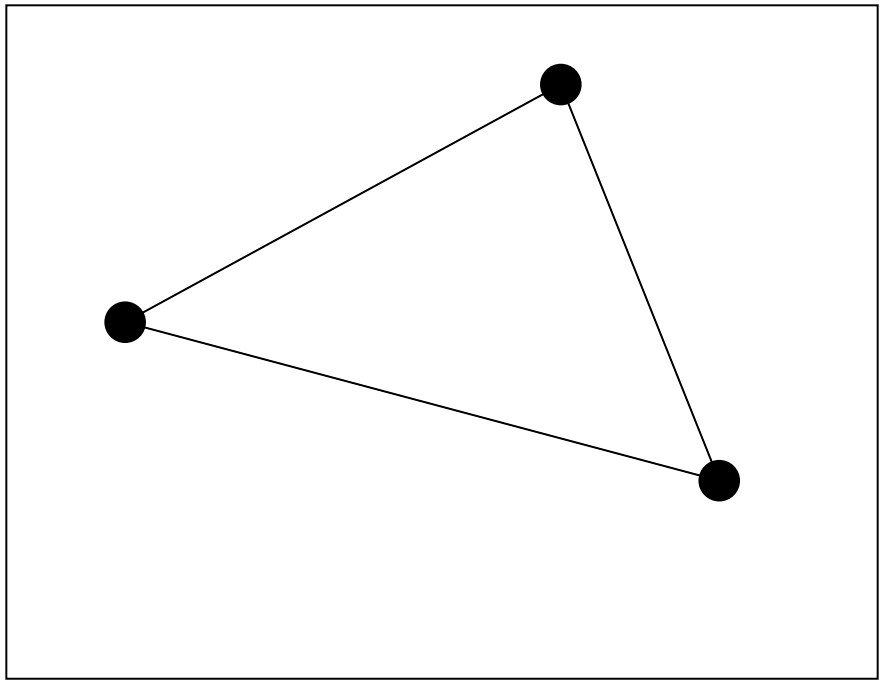




Match clusters to increase modularity.

Agglomerative clustering

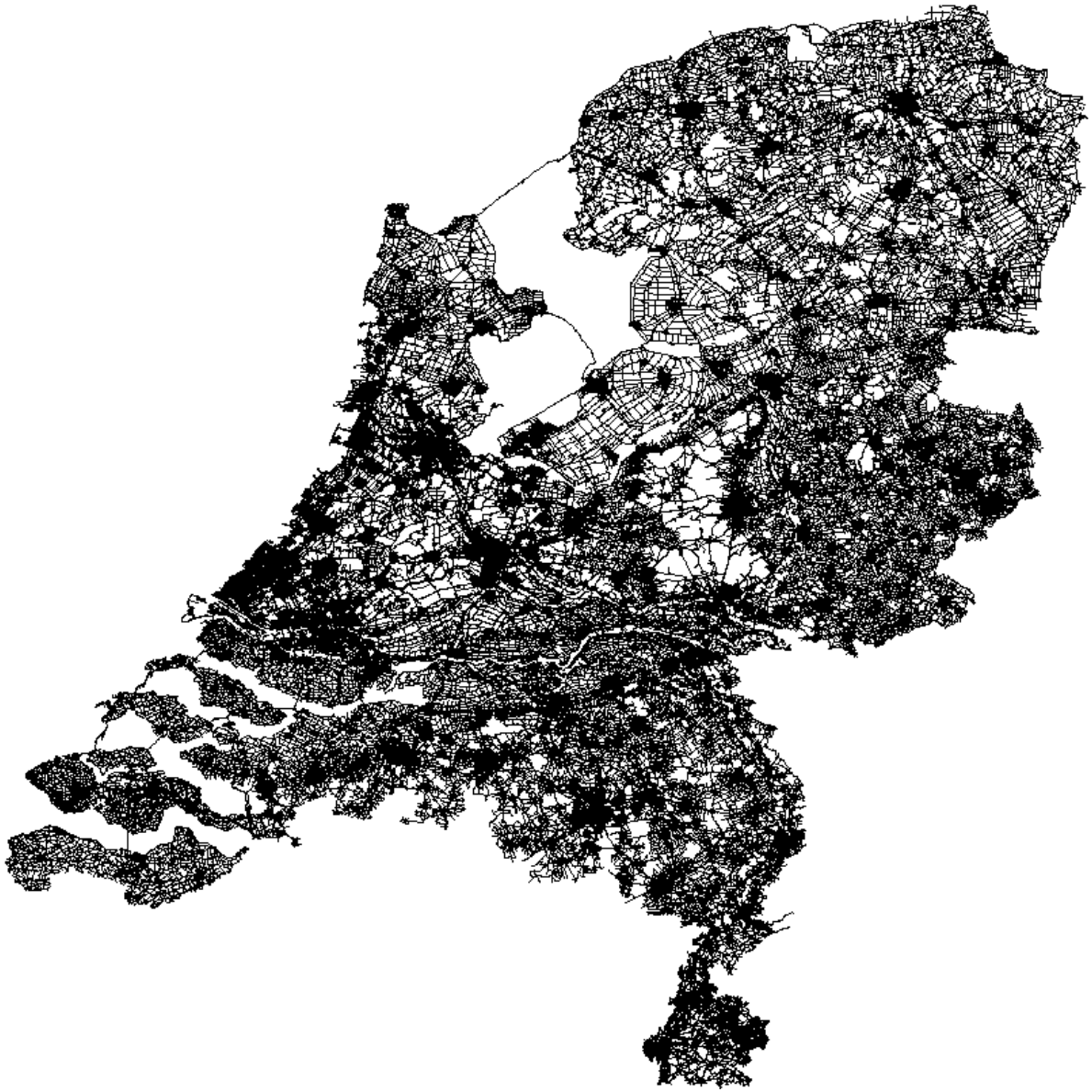




Return clustering with highest modularity.

Agglomerative clustering (netherlands)

0 iterations.



Agglomerative clustering (netherlands)

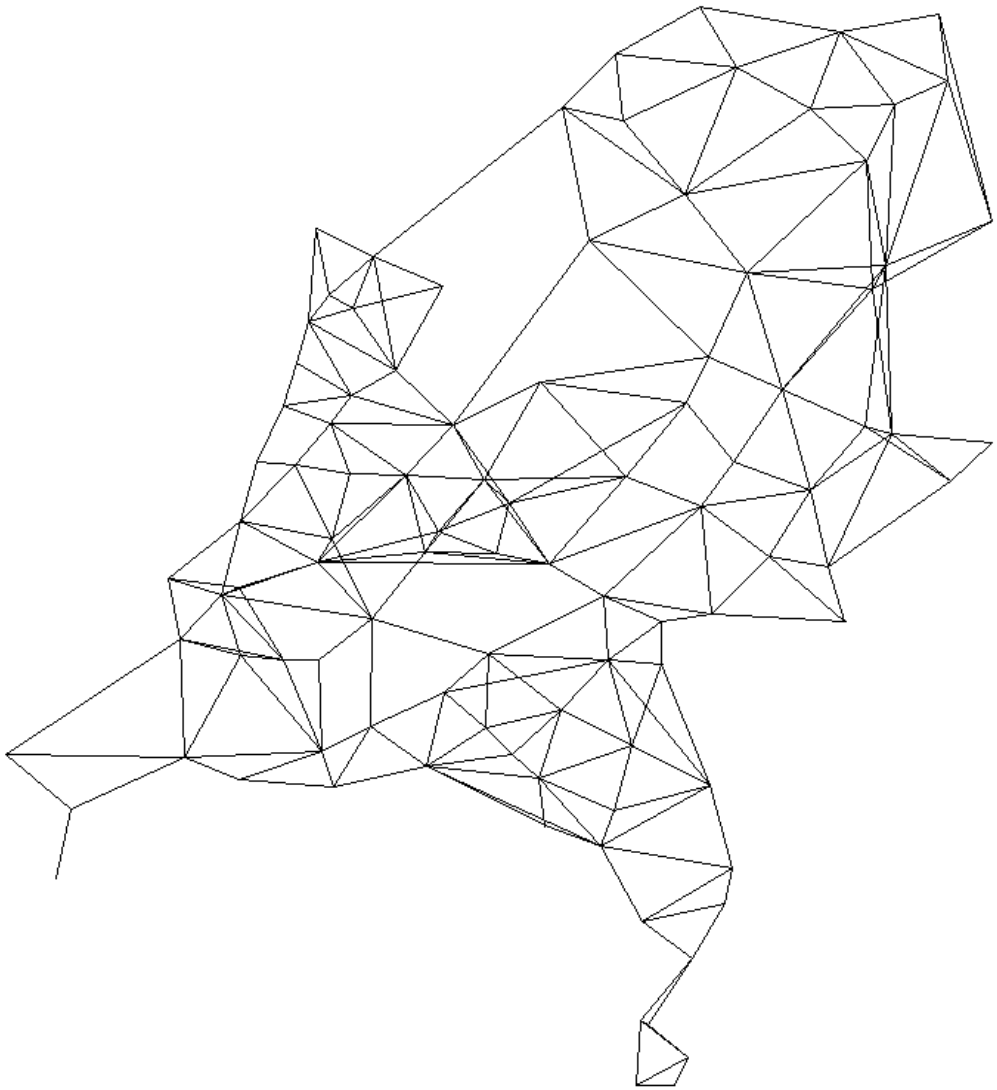
11 iterations.

Agglomerative clustering (netherlands)

21 iterations.

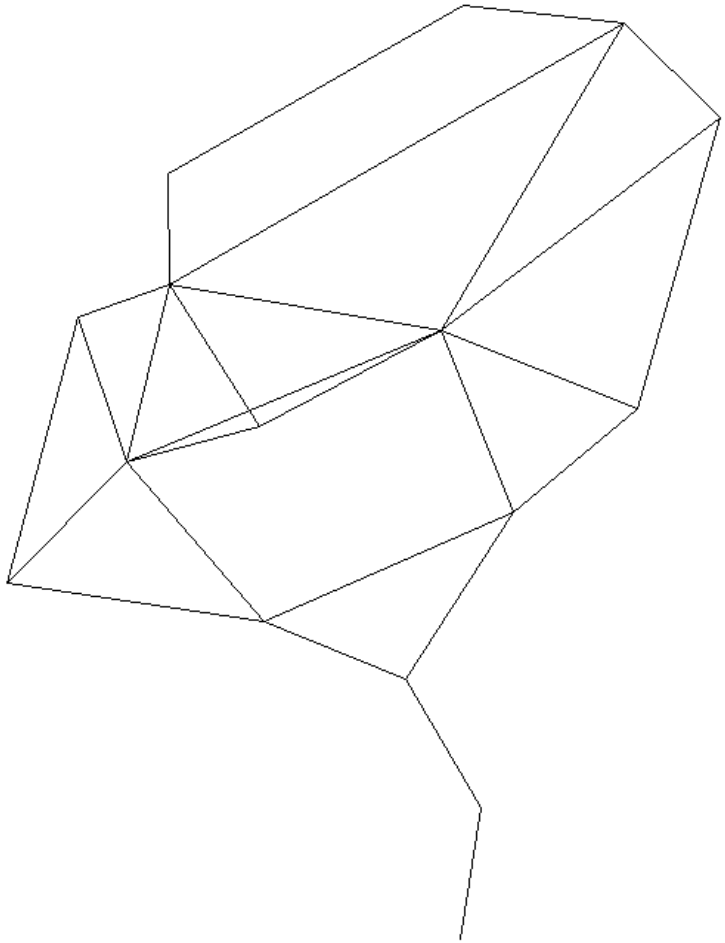
Agglomerative clustering (netherlands)

26 iterations.



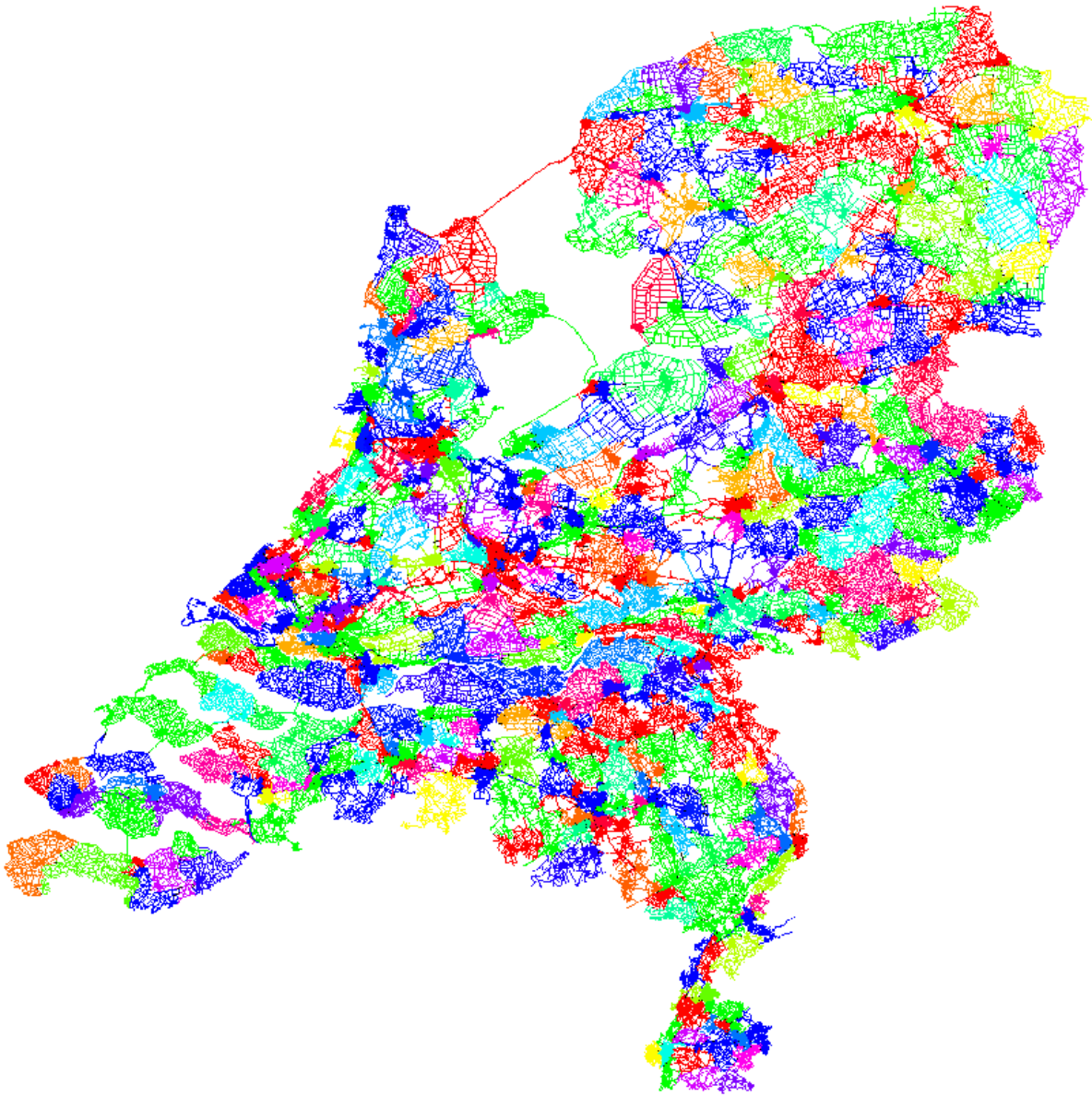
Agglomerative clustering (netherlands)

33 iterations.



Agglomerative clustering (netherlands)

Final clustering.



Parallel agglomerative clustering

1. Start with all vertices being a separate cluster: $\mathcal{C} = \{\{v\} \mid v \in V\}$.
2. Find a heavy matching of clusters with edge weights

$$\frac{1}{2\,\Omega^2}\left(2\,\Omega\,\omega(\mathrm{cut}(C, C')) - \zeta(C)\,\zeta(C')\right).$$

3. Extend matching to avoid getting stuck.
4. Merge all matched clusters, summing ζ and ω.
5. Go to step 2 until only a single cluster remains.
6. Return encountered clustering with highest modularity.

Parallel agglomerative clustering

1. Start with all vertices being a separate cluster: $\mathcal{C} = \{\{v\} \mid v \in V\}$.
2. Find a heavy matching of clusters with edge weights

$$\frac{1}{2\,\Omega^2}\left(2\,\Omega\,\omega(\mathrm{cut}(C, C')) - \zeta(C)\,\zeta(C')\right).$$

3. Extend matching to avoid getting stuck.
4. Merge all matched clusters, summing ζ and ω.
5. Go to step 2 until only a single cluster remains.
6. Return encountered clustering with highest modularity.

We make use of parallelism in steps 2, 3, and 4.

GPU matching problems

- For a graph $G = (V, E)$, a matching is a collection $M \subseteq E$ of edges that are disjoint.

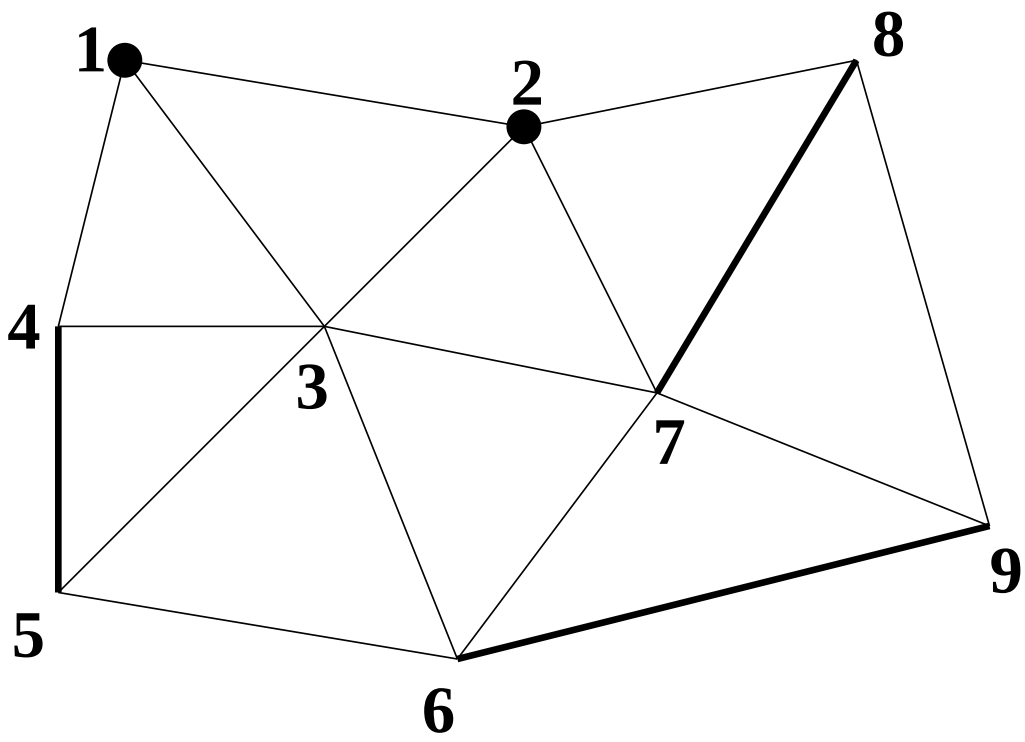
GPU matching problems

- For a graph $G = (V, E)$, a matching is a collection $M \subseteq E$ of edges that are disjoint.
- Performing matching in parallel is problematic.

GPU matching problems

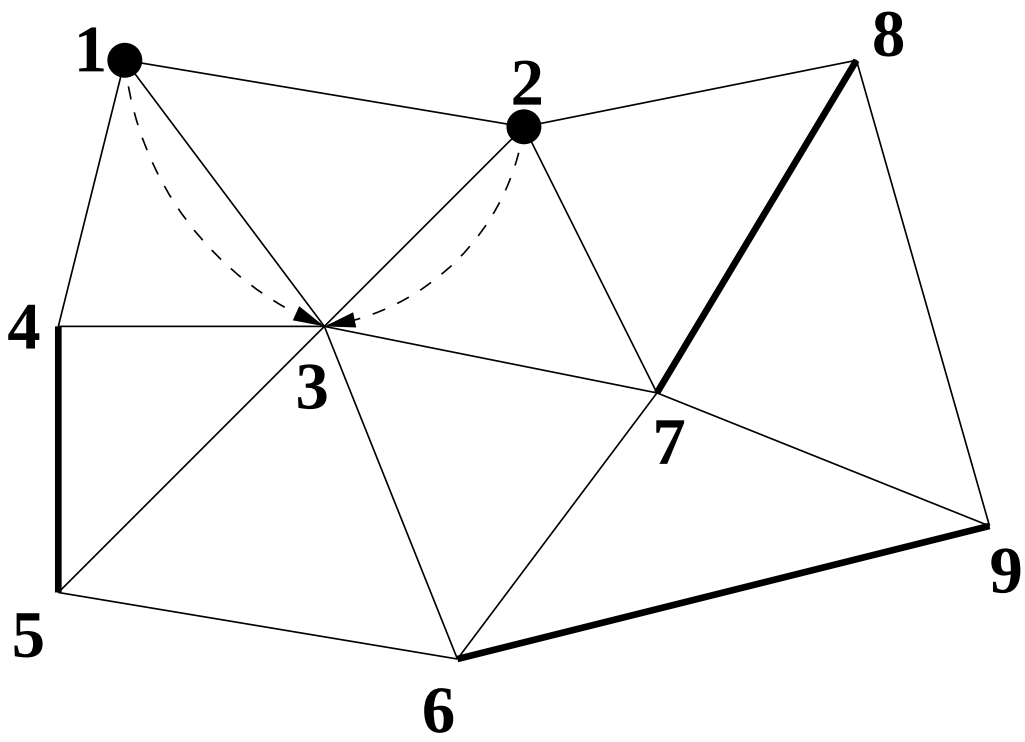
- For a graph $G = (V, E)$, a matching is a collection $M \subseteq E$ of edges that are disjoint.
- Performing matching in parallel is problematic.
- Disjoint edges requirement leads to serialisation.

GPU matching problems



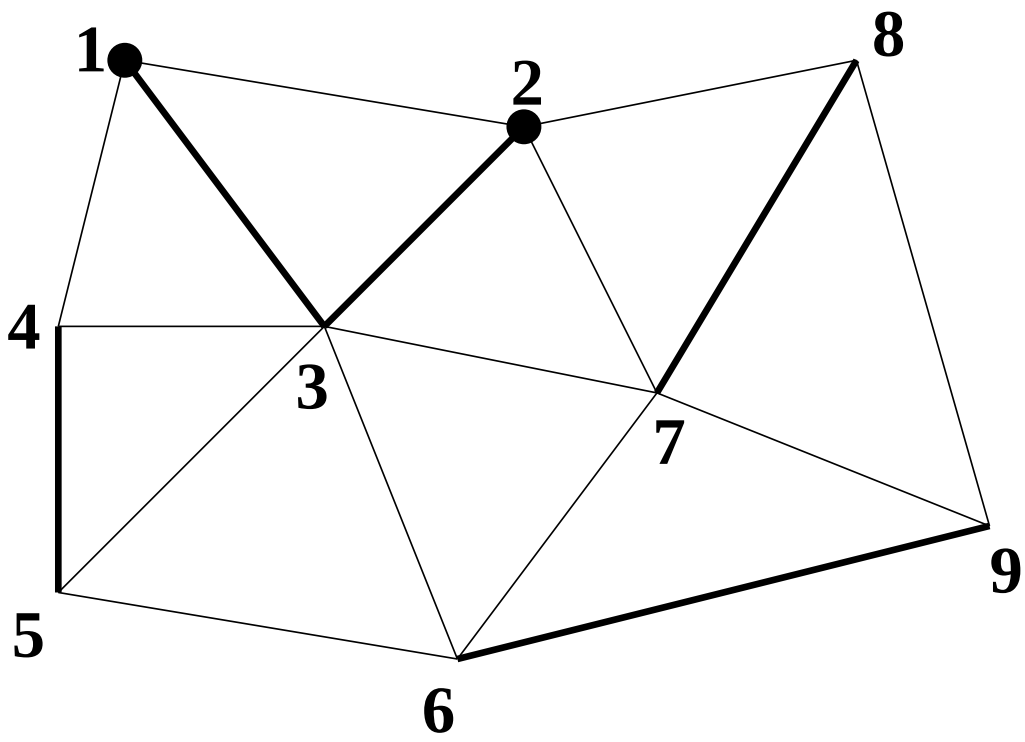
Suppose we match vertices simultaneously.

GPU matching problems



Vertices find an unmatched neighbour. . .

GPU matching problems



. . . but generate an invalid matching.

GPU matching

- To solve this we create two groups of vertices: **blue** and **red**.

GPU matching

- To solve this we create two groups of vertices: **blue** and **red**.
- **Blue** vertices propose.

GPU matching

- To solve this we create two groups of vertices: **blue** and **red**.
- **Blue** vertices propose.
- **Red** vertices respond.

GPU matching

- To solve this we create two groups of vertices: **blue** and **red**.
- **Blue** vertices propose.
- **Red** vertices respond.
- Proposals that were responded to are matched.

GPU matching

- To solve this we create two groups of vertices: **blue** and **red**.
- **Blue** vertices propose.
- **Red** vertices respond.
- Proposals that were responded to are matched.
- Store matching in a map $\pi : V \rightarrow \mathbb{N}$:

$$\{u, v\} \in M \qquad \Longleftrightarrow \qquad \pi(u) = \pi(v).$$

GPU matching (implementation)

- The graph (neighbour ranges, indices, and weights) is stored as a triplet of 1D textures on the GPU.

GPU matching (implementation)

- The graph (neighbour ranges, indices, and weights) is stored as a triplet of 1D textures on the GPU.
- We create one thread for each vertex in V.

GPU matching (implementation)

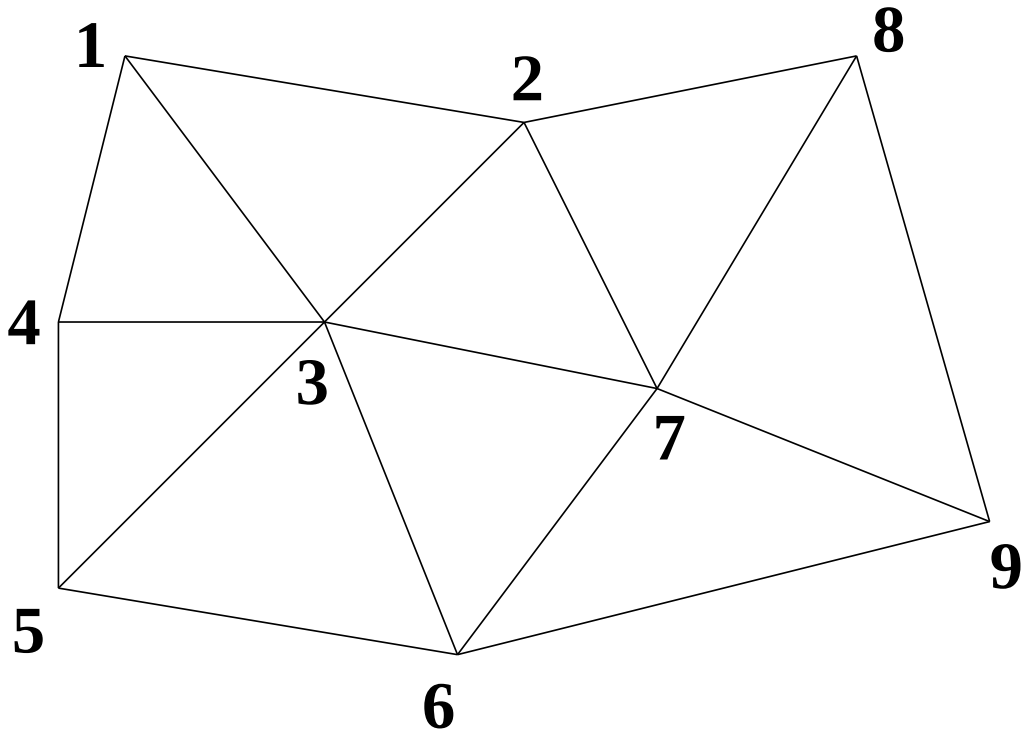
- The graph (neighbour ranges, indices, and weights) is stored as a triplet of 1D textures on the GPU.
- We create one thread for each vertex in V.
- Each vertex $v \in V$ only updates
  - its colour/matching value $\pi(v)$;
  - and its proposal/response value $\sigma(v)$.

GPU matching (implementation)

- The graph (neighbour ranges, indices, and weights) is stored as a triplet of 1D textures on the GPU.
- We create one thread for each vertex in V.
- Each vertex $v \in V$ only updates
  - its colour/matching value $\pi(v)$;
  - and its proposal/response value $\sigma(v)$.
- Both π and σ are stored in 1D arrays in global memory.

GPU matching (algorithm)

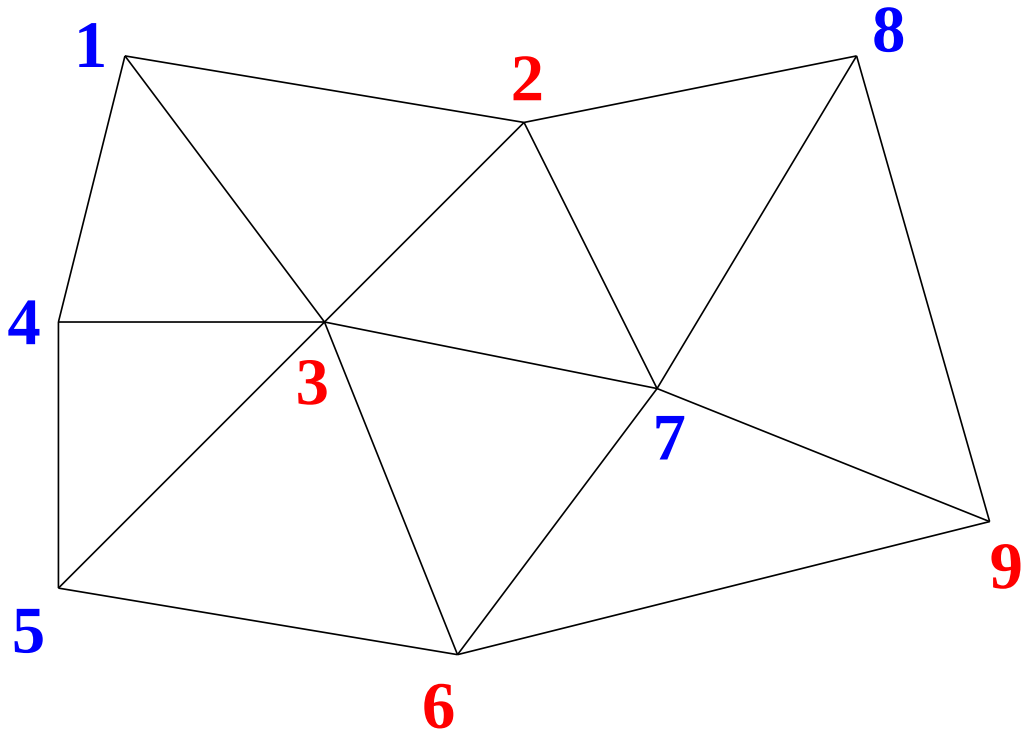
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	-	-	-	-	-	-	-	-	-
σ	-	-	-	-	-	-	-	-	-

GPU matching (algorithm)

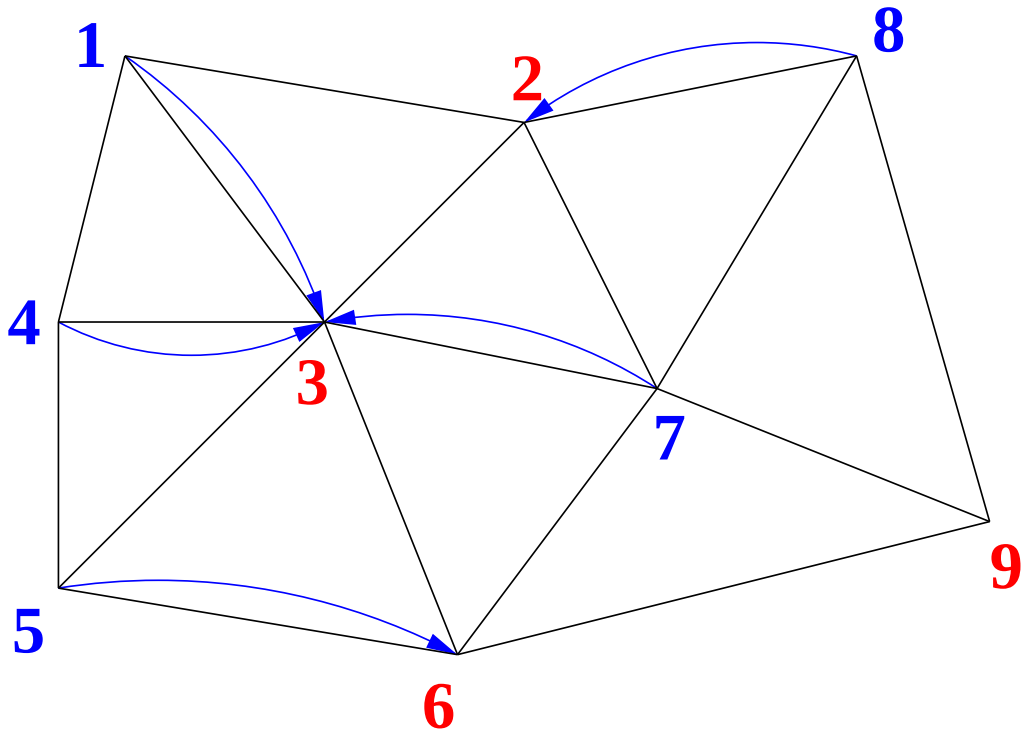
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**b**	**r**	**r**	**b**	**b**	**r**	**b**	**b**	**r**
σ	-	-	-	-	-	-	-	-	-

GPU matching (algorithm)

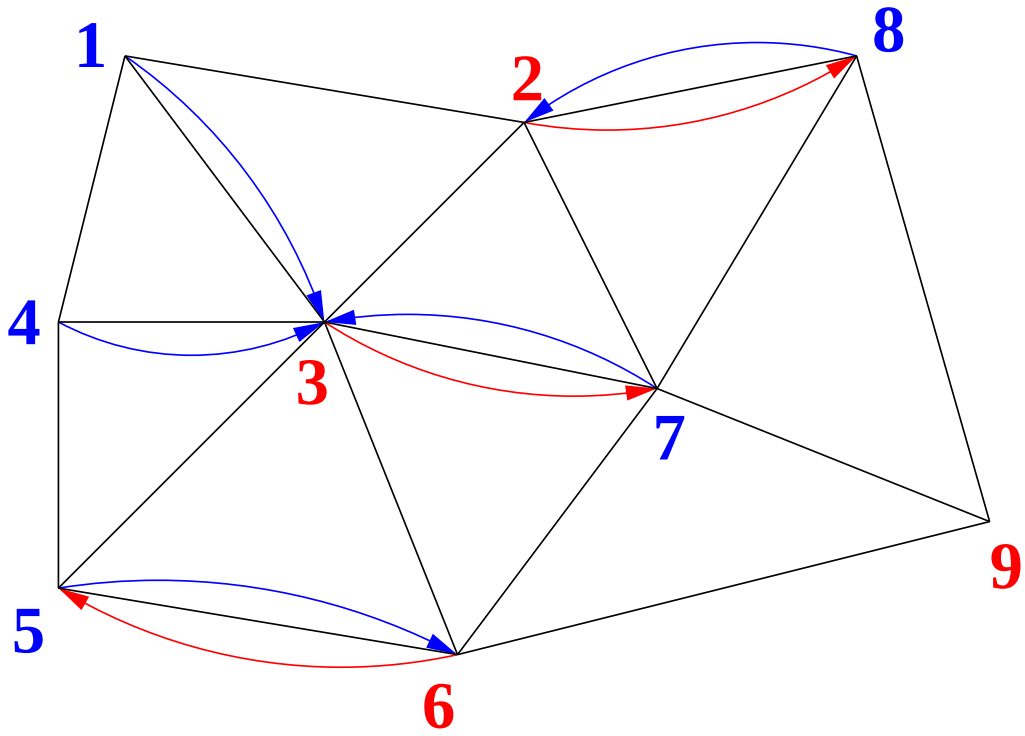
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**b**	**r**	**r**	**b**	**b**	**r**	**b**	**b**	**r**
σ	3	-	-	3	6	-	3	2	-

GPU matching (algorithm)

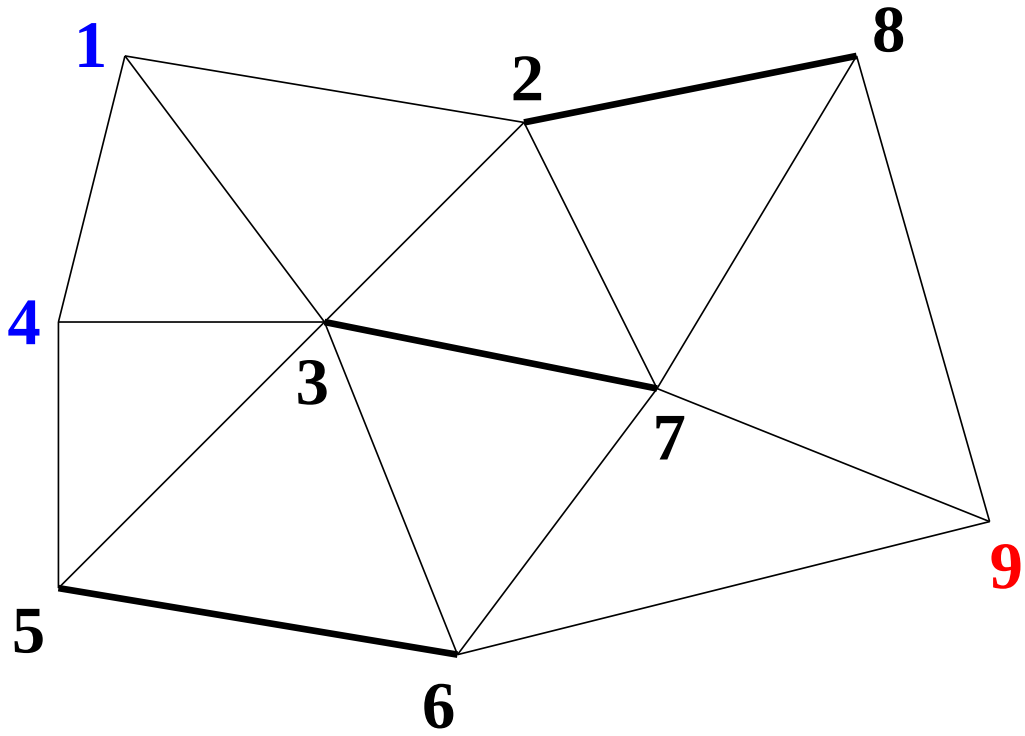
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**b**	**r**	**r**	**b**	**b**	**r**	**b**	**b**	**r**
σ	3	8	7	3	6	5	3	2	-

GPU matching (algorithm)

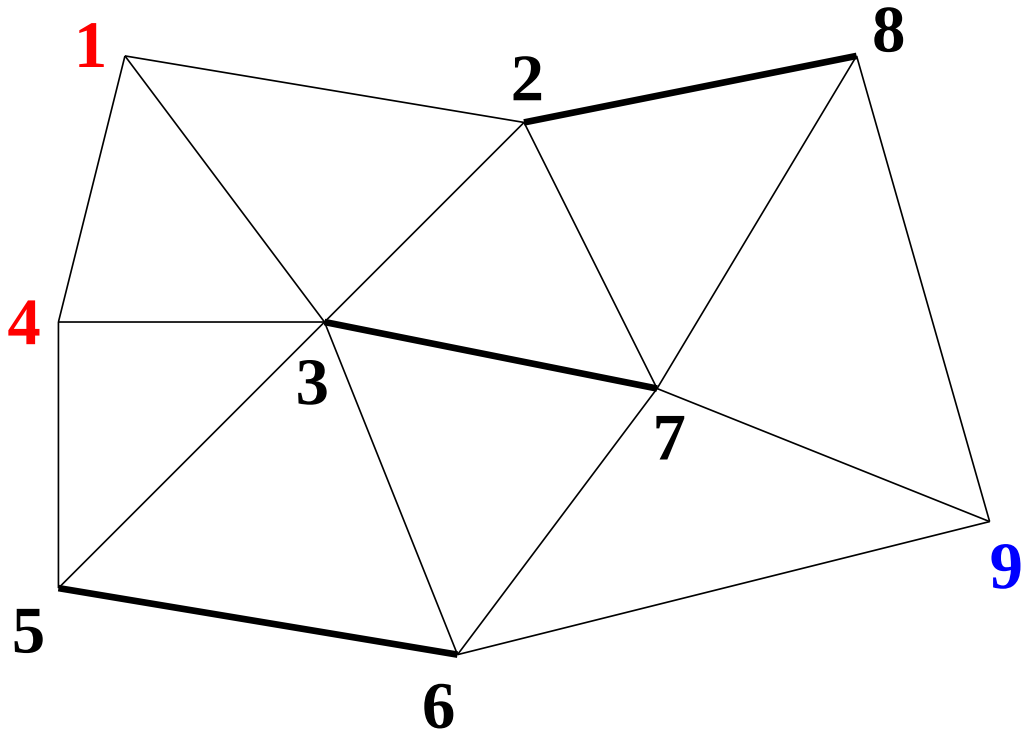
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**b**	2	3	**b**	5	5	3	2	**r**
σ	3	8	7	3	6	5	3	2	-

GPU matching (algorithm)

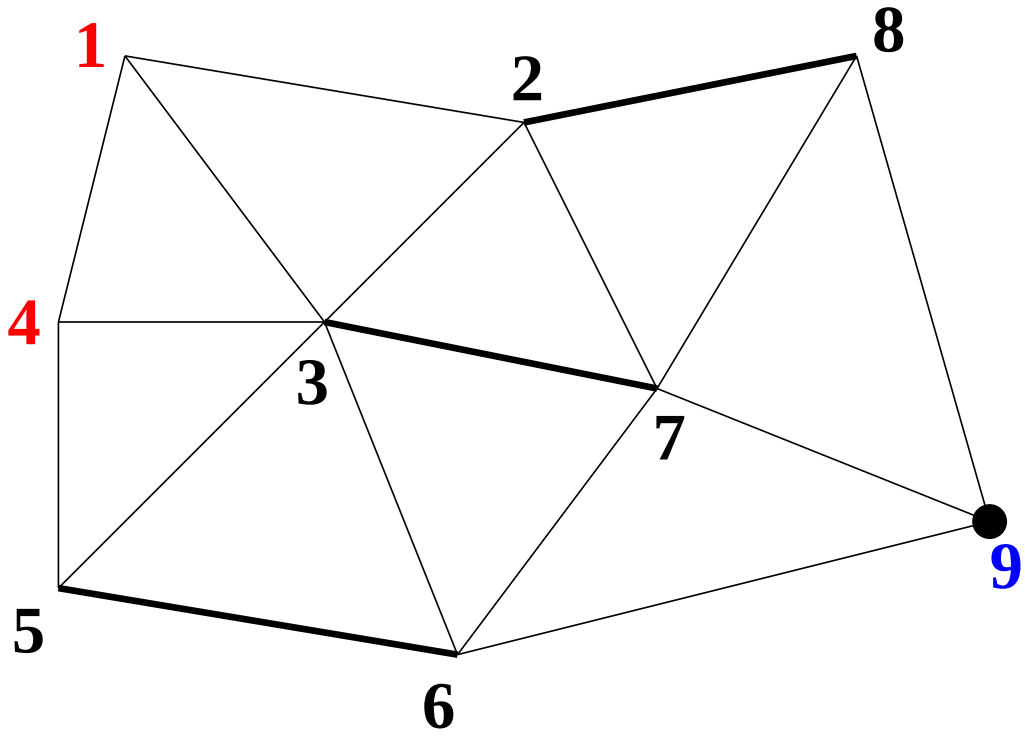
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**r**	2	3	**r**	5	5	3	2	**b**
σ	3	8	7	3	6	5	3	2	-

GPU matching (algorithm)

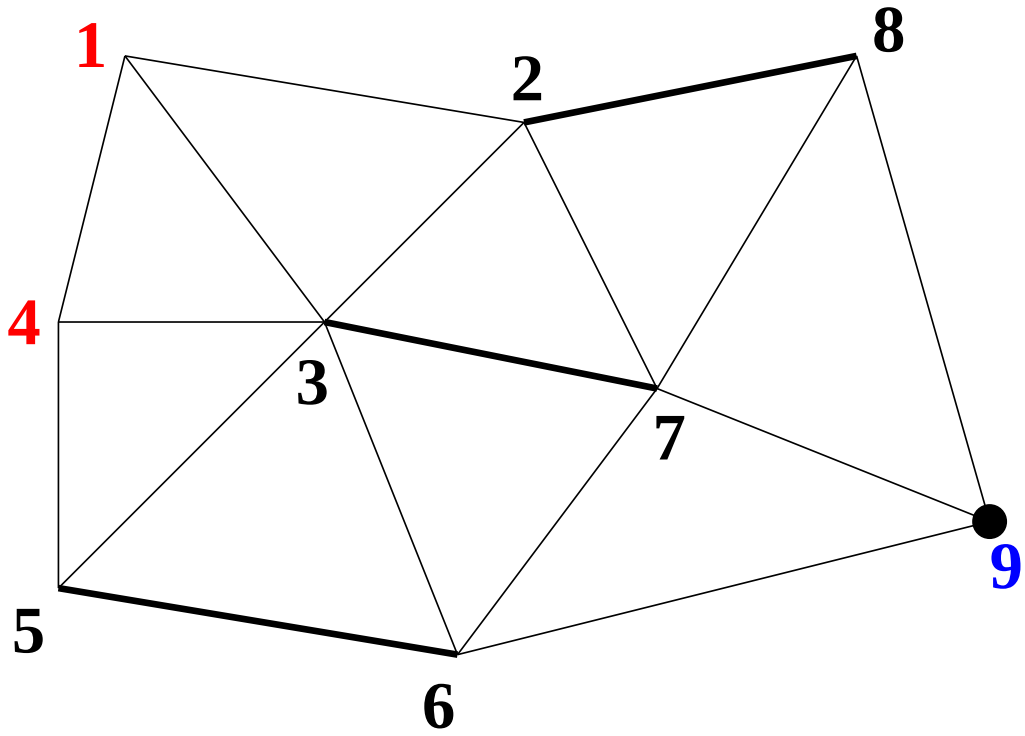
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**r**	2	3	**r**	5	5	3	2	**b**
σ	-	-	-	-	-	-	-	-	**d**

GPU matching (algorithm)

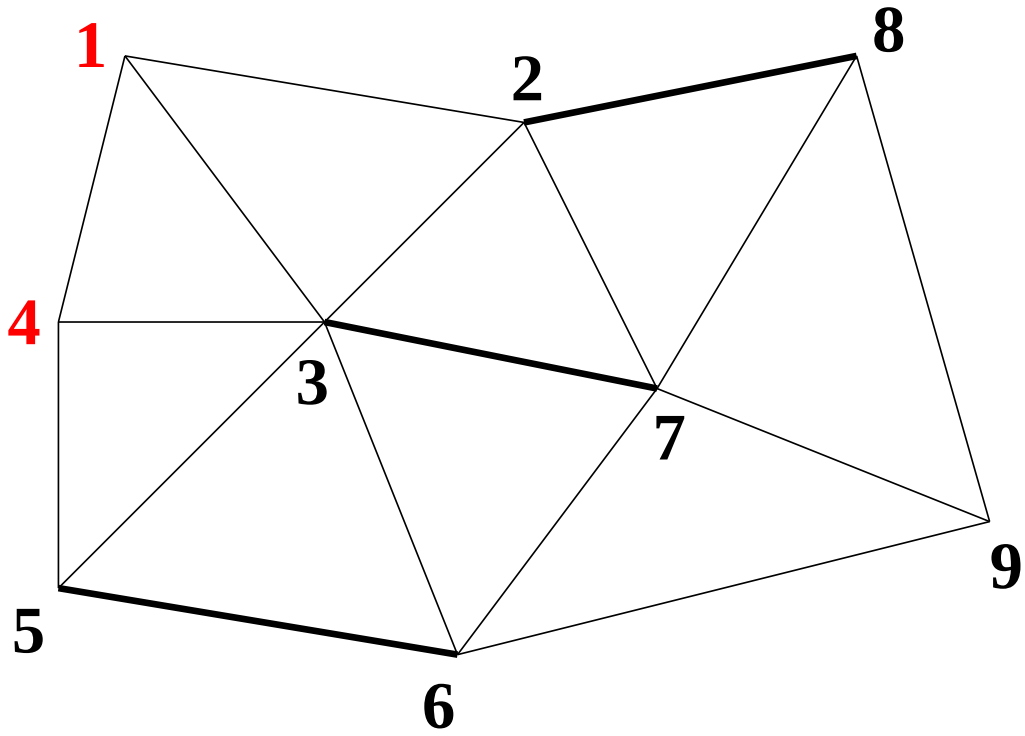
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**r**	2	3	**r**	5	5	3	2	**b**
σ	-	-	-	-	-	-	-	-	**d**

GPU matching (algorithm)

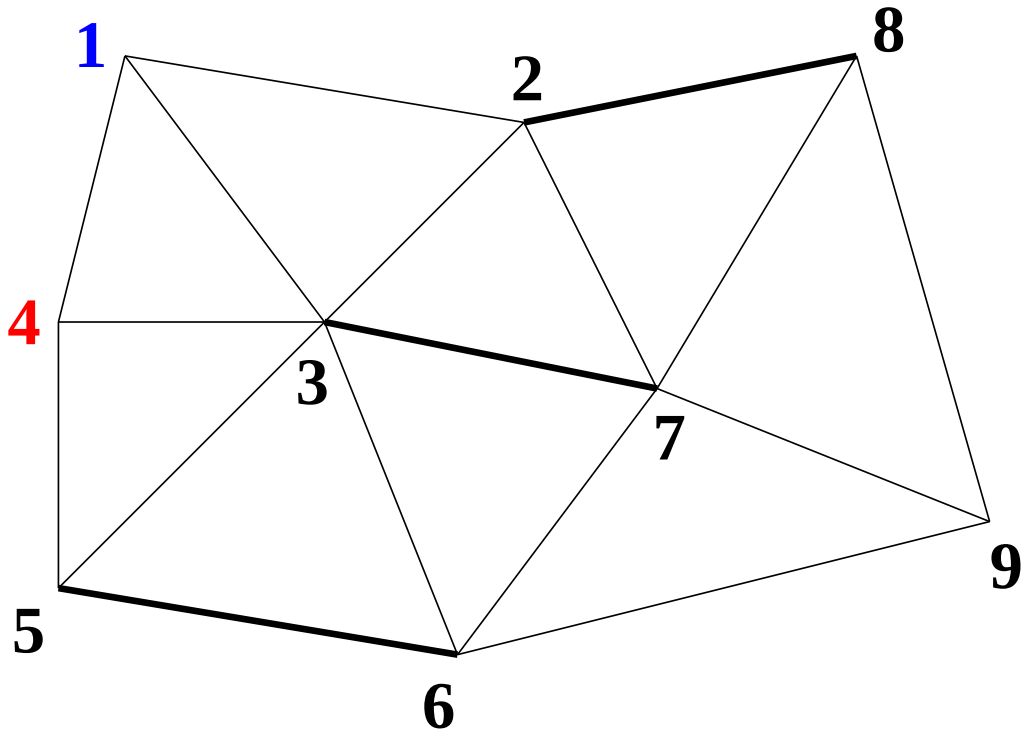
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**r**	2	3	**r**	5	5	3	2	**d**
σ	-	-	-	-	-	-	-	-	**d**

GPU matching (algorithm)

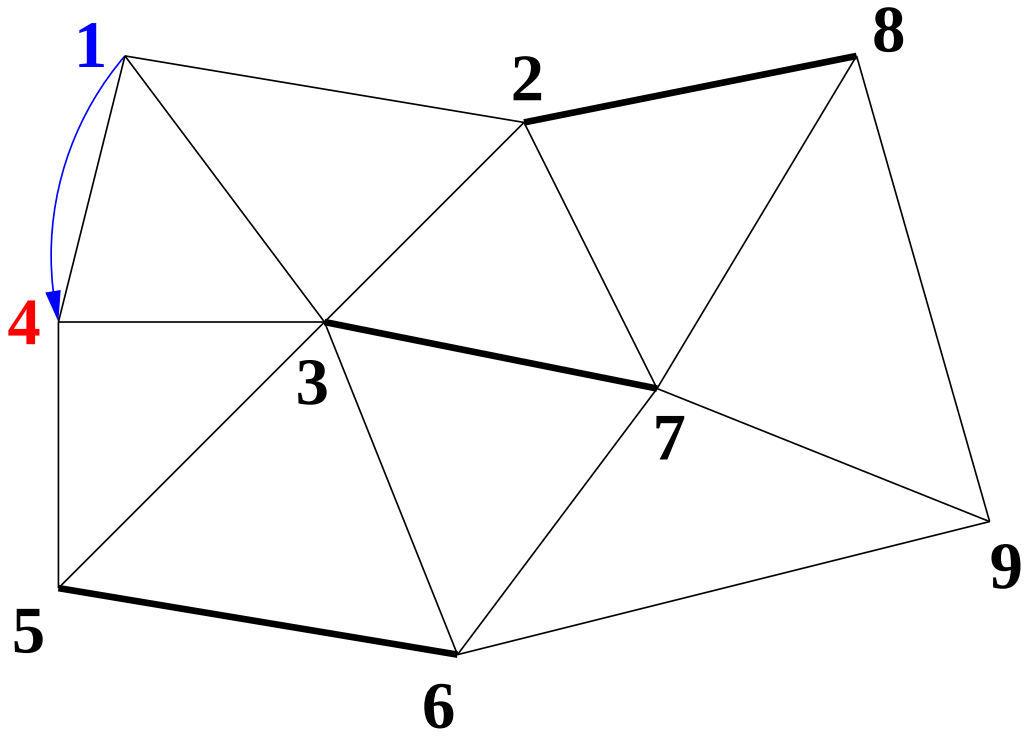
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**b**	2	3	**r**	5	5	3	2	**d**
σ	-	-	-	-	-	-	-	-	**d**

GPU matching (algorithm)

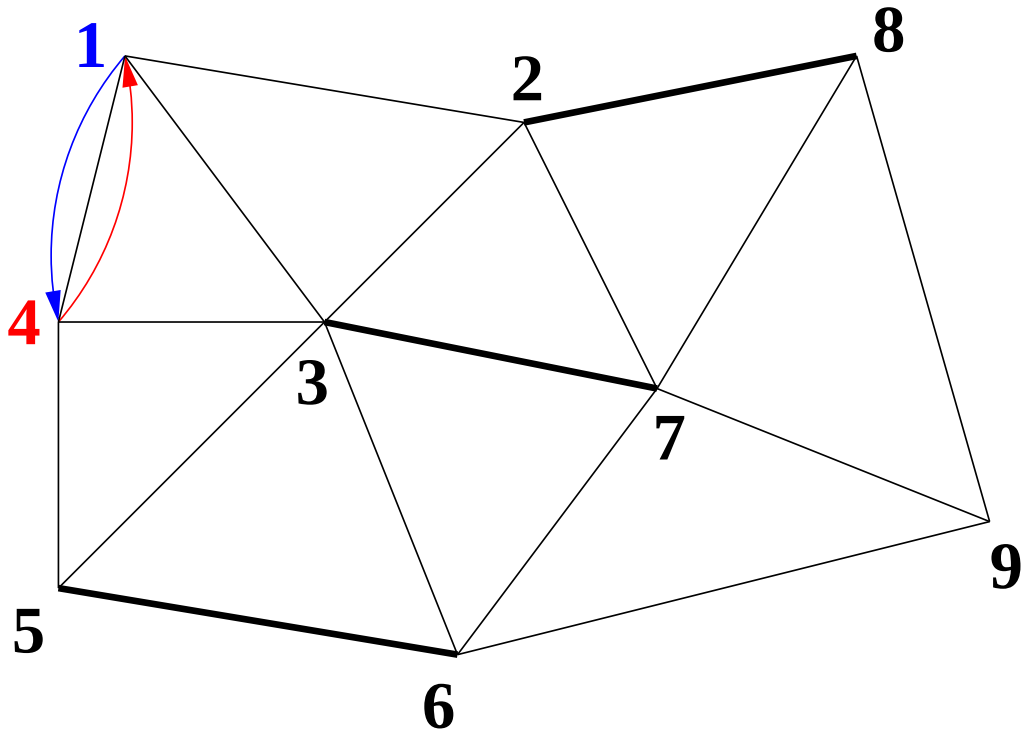
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**b**	2	3	**r**	5	5	3	2	**d**
σ	4	-	-	-	-	-	-	-	-

GPU matching (algorithm)

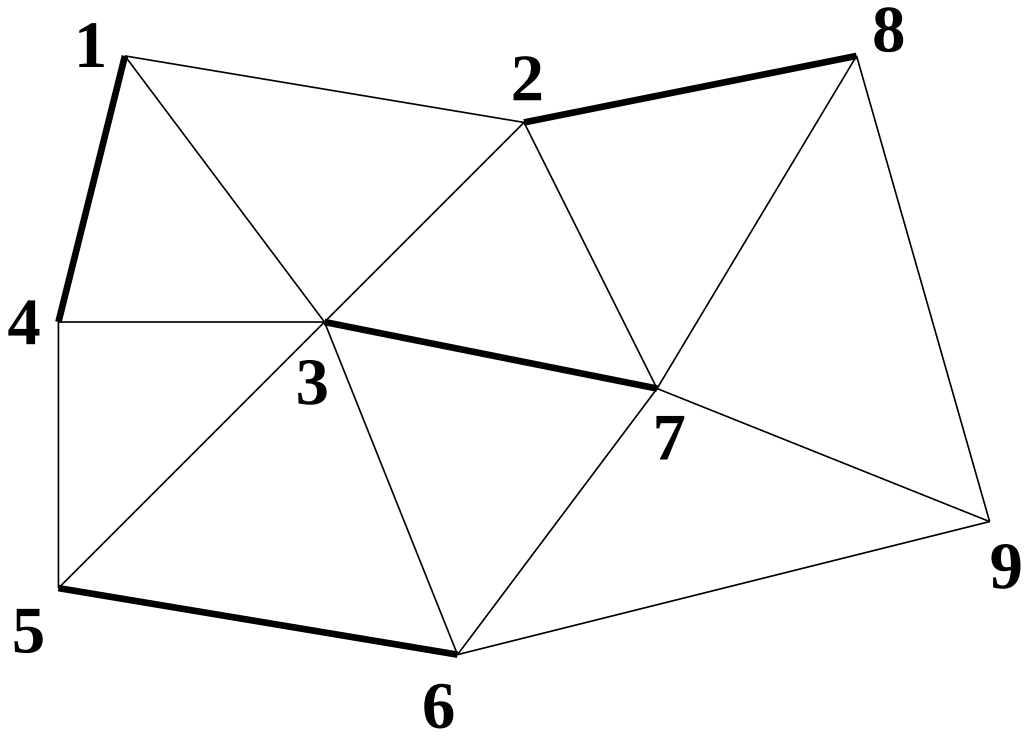
Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	**b**	2	3	**r**	5	5	3	2	**d**
σ	4	-	-	1	-	-	-	-	-

GPU matching (algorithm)

Colour
Propose
Respond
Match



	1	2	3	4	5	6	7	8	9
π	1	2	3	1	5	5	3	2	**d**
σ	4	-	-	1	-	-	-	-	-

Results

- Created an implementation on the GPU using CUDA and on the CPU using TBB.

Results

- Created an implementation on the GPU using CUDA and on the CPU using TBB.
- Results are averaged over 16 runs.

Results

- Created an implementation on the GPU using CUDA and on the CPU using TBB.
- Results are averaged over 16 runs.
- Time: matching and CPU $\leftrightarrow$ GPU data transfer, not disk I/O.

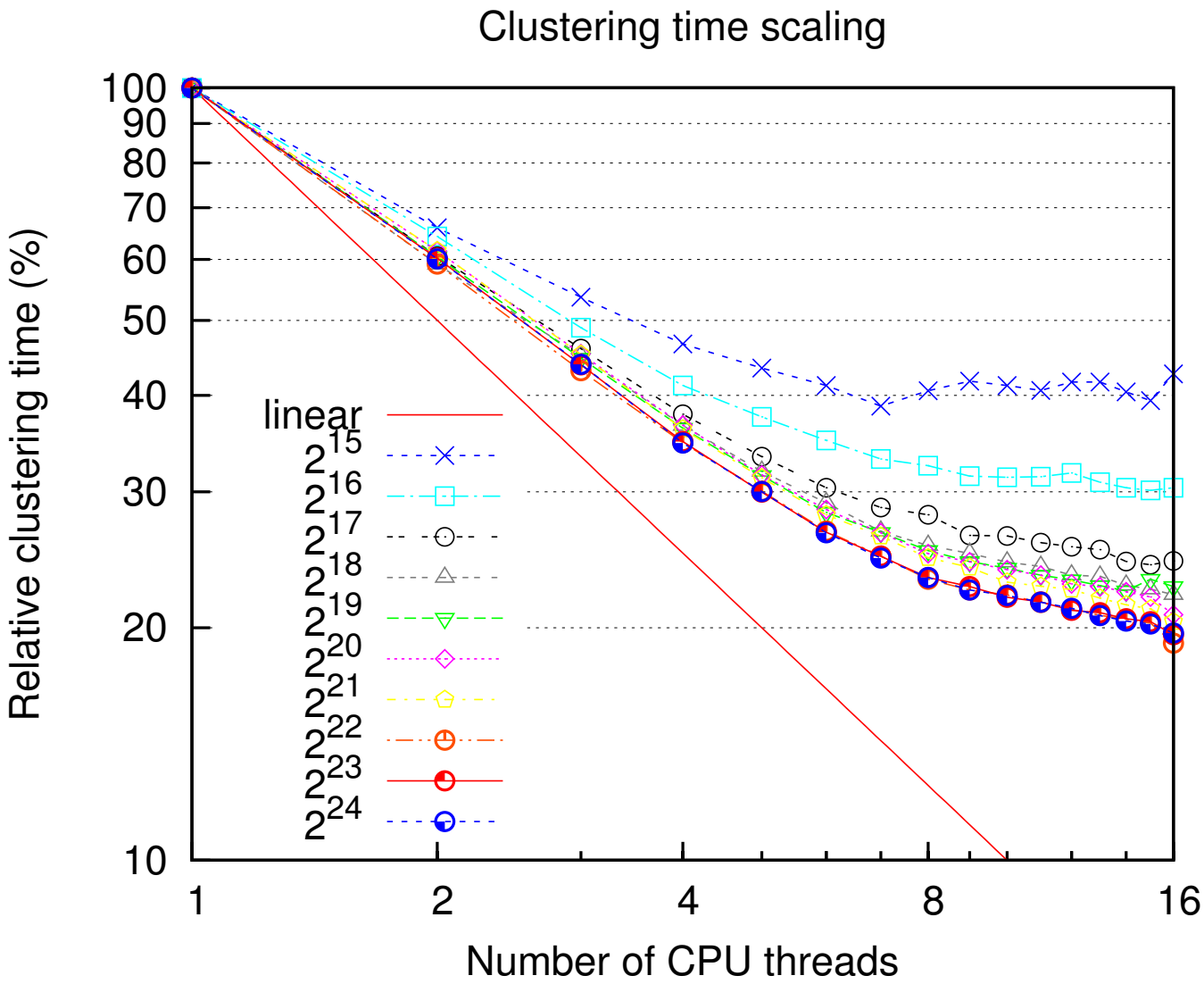
Results

- Created an implementation on the GPU using CUDA and on the CPU using TBB.
- Results are averaged over 16 runs.
- Time: matching and CPU $\leftrightarrow$ GPU data transfer, not disk I/O.
- Test set: 10th DIMACS challenge.

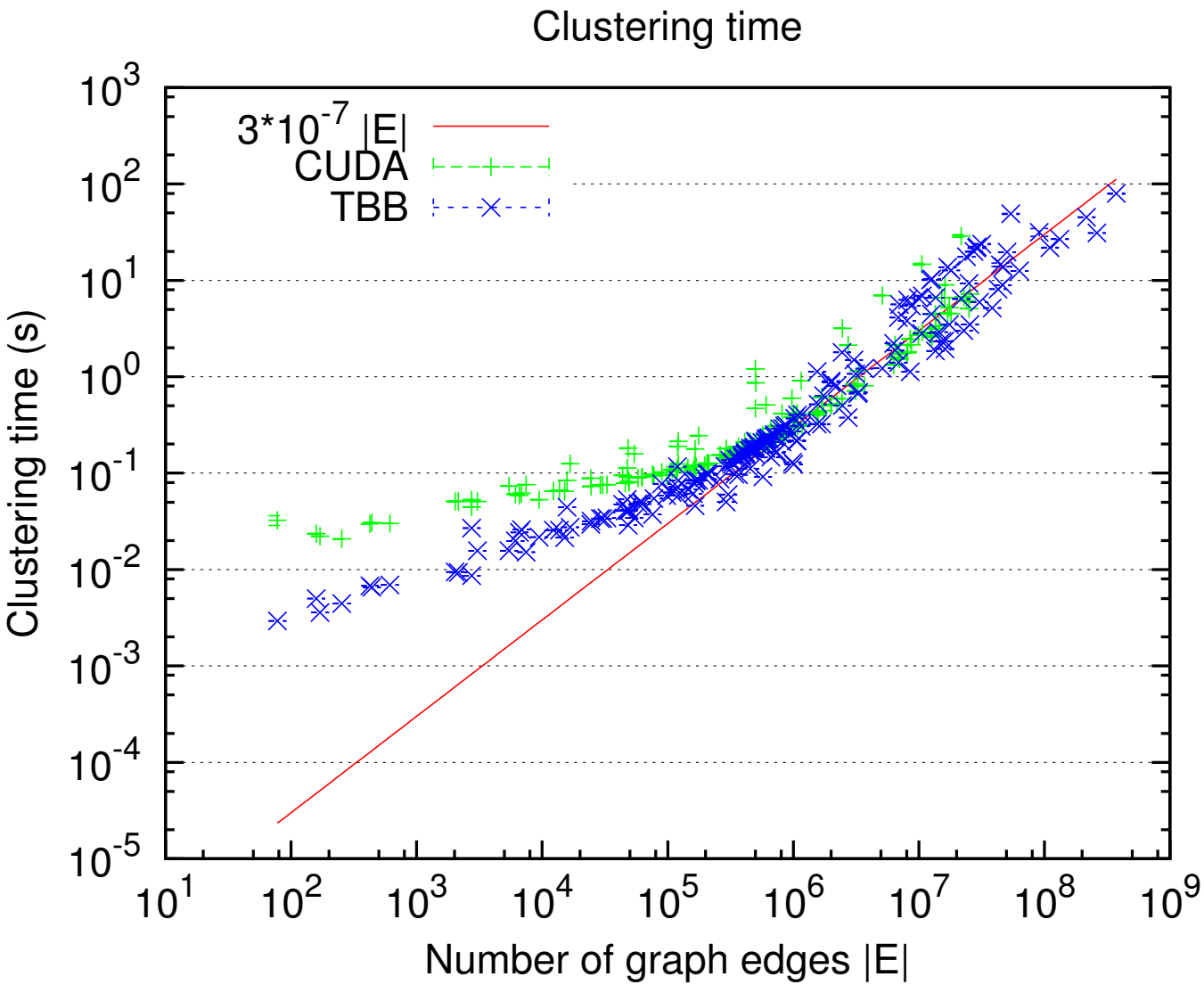
Results

- Created an implementation on the GPU using CUDA and on the CPU using TBB.
- Results are averaged over 16 runs.
- Time: matching and CPU $\leftrightarrow$ GPU data transfer, not disk I/O.
- Test set: 10th DIMACS challenge.
- Test hardware: dual quad-core Xeon E5620 and an NVIDIA Tesla C2050 (thanks: the Little Green Machine project).

Results (scaling)



Results (time)



Results (quality)

	$\|V\|$	$\|E\|$	CUDA	TBB	Ovelgönne et al. (2010)
karate	34	78	0.363	0.383	0.412
jazz	198	2,742	0.314	0.369	0.444
email	1,133	5,451	0.440	0.473	0.572
PGP	10,680	24,316	0.809	0.841	0.880

Results (quality)

	$\|V\|$	$\|E\|$	CUDA	TBB	Ovelgönne et al. (2010)
`karate`	34	78	0.363	0.383	0.412
`jazz`	198	2,742	0.314	0.369	0.444
`email`	1,133	5,451	0.440	0.473	0.572
`PGP`	10,680	24,316	0.809	0.841	0.880

- Lower quality, because we do not use local refinement.

Results (time)

- Our algorithm is very fast for large graphs.

Results (time)

- Our algorithm is very fast for large graphs.
- CUDA: `road_central`, $|V| = 14{,}081{,}816$, $|E| = 16{,}933{,}413$, modularity 0.996 clustering in 4.6 seconds.

Results (time)

- Our algorithm is very fast for large graphs.
- CUDA: `road_central`, $|V| = 14{,}081{,}816$, $|E| = 16{,}933{,}413$, modularity 0.996 clustering in 4.6 seconds.
- TBB: `uk-2002`, $|V| = 18{,}520{,}486$, $|E| = 261{,}787{,}258$, modularity 0.974 clustering in 31 seconds.

Conclusion

- We presented a fine-grained shared-memory parallel clustering algorithm.

Conclusion

- We presented a fine-grained shared-memory parallel clustering algorithm.
- This algorithm is suitable for both multi-core CPUs and GPUs.

Conclusion

- We presented a fine-grained shared-memory parallel clustering algorithm.
- This algorithm is suitable for both multi-core CPUs and GPUs.
- The algorithm is very fast, but quality could be improved by parallel local refinement.

Questions

$\exists$ any questions?

Modularity

- How to measure clustering quality?

Modularity

- How to measure clustering quality?
- Let $\mathcal{C} = \{C_1, \ldots, C_k\}$ and define

$$e_{ij} := \text{weighted fraction of edges between } C_i \text{ and } C_j.$$

Modularity

- How to measure clustering quality?
- Let $\mathcal{C} = \{C_1, \ldots, C_k\}$ and define

$$e_{i\,j} := \text{weighted fraction of edges between } C_i \text{ and } C_j.$$

- Maximise intra-cluster edges:

$$\max_{\mathcal{C}} \left(\sum_{i=1}^{k} e_{i\,i} \right).$$

Modularity

- How to measure clustering quality?
- Let $\mathcal{C} = \{C_1, \ldots, C_k\}$ and define

$$e_{i\,j} := \text{weighted fraction of edges between } C_i \text{ and } C_j.$$

- Maximise intra-cluster edges:

$$\max_{\mathcal{C}} \left(\sum_{i=1}^{k} e_{i\,i} \right).$$

- But this leads to $\mathcal{C} = \{V\}$ as optimal solution!

Modularity

- Define

$$a_i := \sum_{j=1}^{k} e_{ij} = \text{weighted fraction of edges incident to } C_i.$$

Modularity

- Define

$$a_i := \sum_{j=1}^{k} e_{ij} = \text{weighted fraction of edges incident to } C_i.$$

- Cut all edges and reconnect vertices randomly, then

$$e_{ij} \approx a_i \, a_j.$$

Modularity

- Define

$$a_i := \sum_{j=1}^{k} e_{i\,j} = \text{weighted fraction of edges incident to } C_i.$$

- Cut all edges and reconnect vertices randomly, then

$$e_{i\,j} \approx a_i\, a_j.$$

- Modularity measures how much better we do than the random case:

$$\text{mod}(\mathcal{C}) := \sum_{i=1}^{k} \left(e_{i\,i} - a_i^2 \right).$$

Modularity

- For any clustering $\mathcal{C}$:

$$-\frac{1}{2} \leq \text{mod}(\mathcal{C}) \leq 1.$$

Modularity

- For any clustering $\mathcal{C}$:

$$-\frac{1}{2} \leq \text{mod}(\mathcal{C}) \leq 1.$$

- Maximising modularity is strongly NP-complete (Brandes et al., 2008).

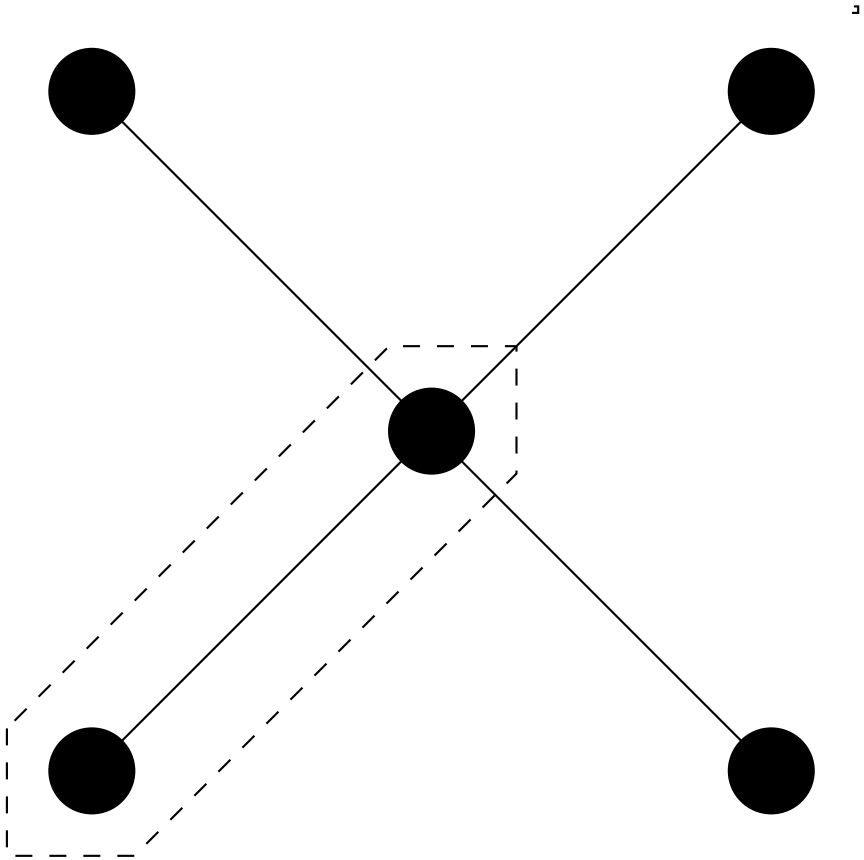
Modularity

- For any clustering $\mathcal{C}$:

$$-\frac{1}{2} \leq \text{mod}(\mathcal{C}) \leq 1.$$

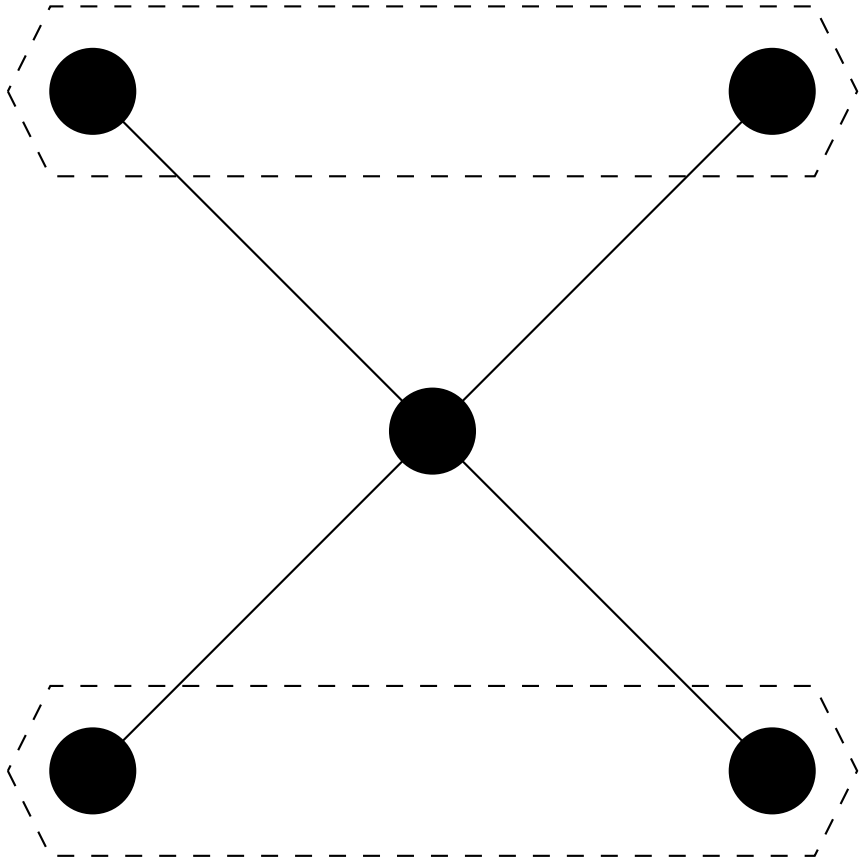
- Maximising modularity is strongly NP-complete (Brandes et al., 2008).
- Modularity maximisation fails to resolve small communities (Kumpula et al., 2007).

Star graphs



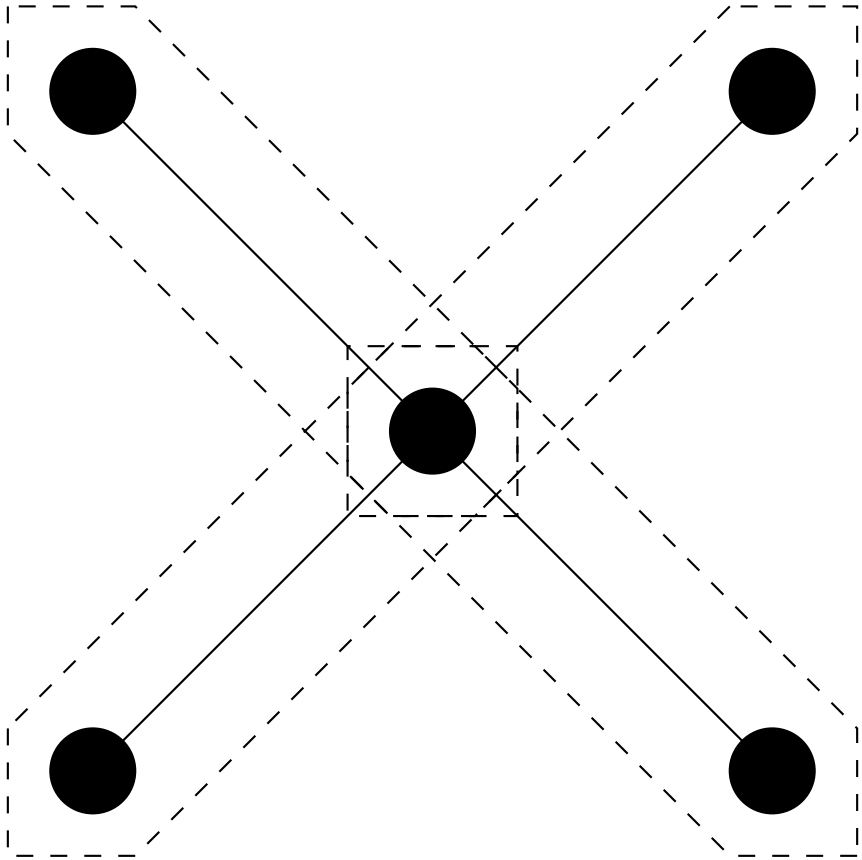
Agglomerative clustering slows down on star graphs.

Star graphs



Merging vertices with the same neighbours is bad for clustering.

Star graphs



So we merge multiple satellites to the same centre.

Centre potential

- To identify star centres and satellites, we propose a centre potential.

Centre potential

- To identify star centres and satellites, we propose a centre potential.
- This potential is defined for vertices v as

$$\mathrm{cp}(v) := \frac{\deg(v)^2}{\sum_{\{u,v\}\in E} \deg(u)}.$$

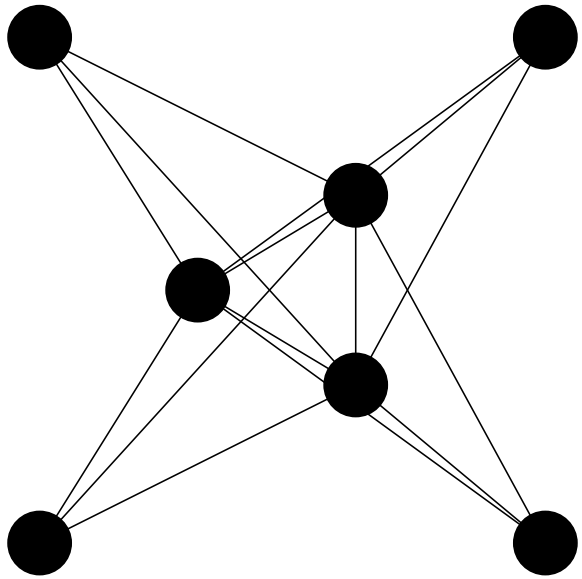
Centre potential

- To identify star centres and satellites, we propose a centre potential.
- This potential is defined for vertices v as

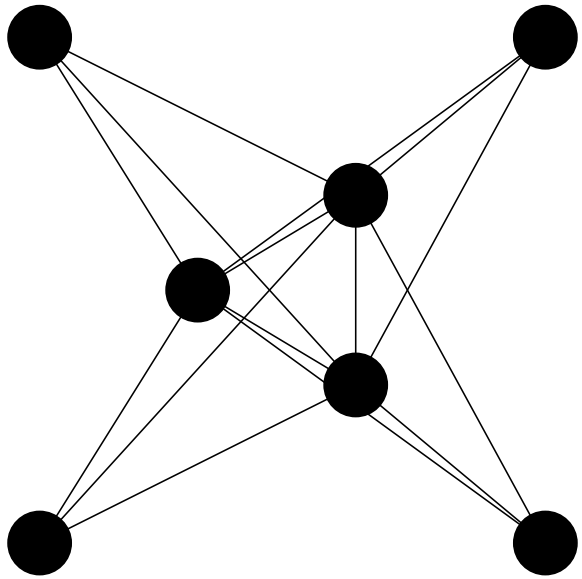
$$\mathrm{cp}(v) := \frac{\deg(v)^2}{\sum_{\{u,v\}\in E} \deg(u)}.$$

- We use $\mathrm{cp}(\cdot)$ to identify satellites and match these to centres.

Centre potential

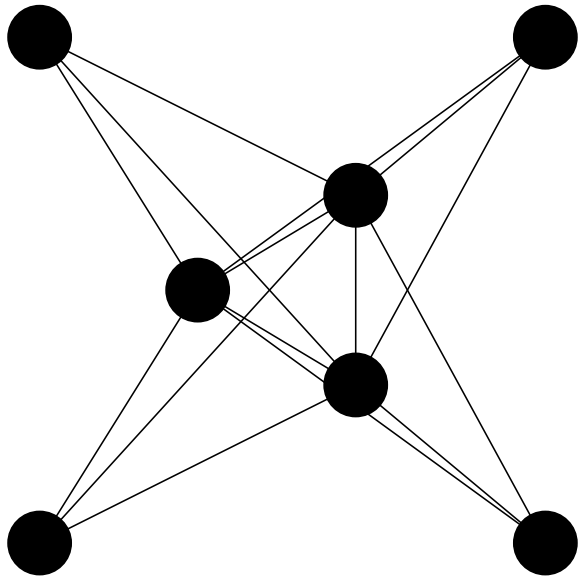


Centre potential



- For a star graph where k satellites are connected to a clique of l vertices with $0 < l < k$, we have that

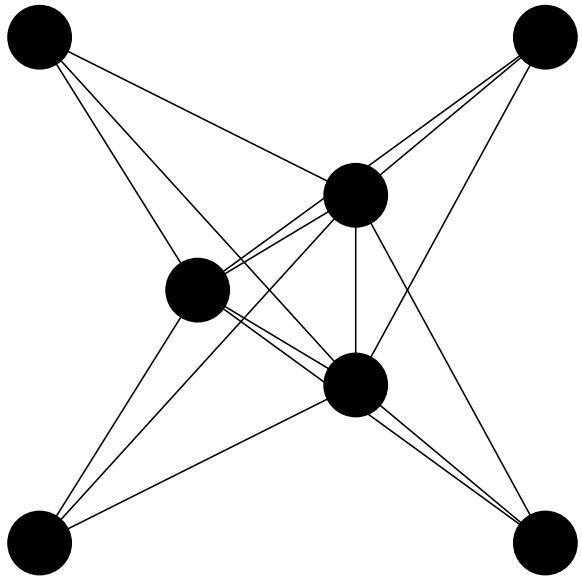
Centre potential



- For a star graph where k satellites are connected to a clique of l vertices with $0 < l < k$, we have that

$$\text{cp(satellite)} \leq \frac{1}{2} \qquad \text{and cp(satellite)} \to 0 \text{ as } k \to \infty,$$

Centre potential



- For a star graph where k satellites are connected to a clique of l vertices with $0 < l < k$, we have that

$$\text{cp(satellite)} \leq \frac{1}{2} \qquad \text{and cp(satellite)} \to 0 \text{ as } k \to \infty,$$

$$\text{cp(centre)} \geq \frac{4}{3} \qquad \text{and cp(centre)} \to \infty \text{ as } k \to \infty.$$

Colouring vertices

- To colour vertices $v \in V$, we use for a fixed $p \in [0, 1]$

$$\pi(v) = \begin{cases} \textbf{blue} & \text{with probability } p, \\ \textbf{red} & \text{with probability } 1 - p. \end{cases}$$

Colouring vertices

- To colour vertices $v \in V$, we use for a fixed $p \in [0, 1]$

$$\pi(v) = \begin{cases} \textbf{blue} & \text{with probability } p, \\ \textbf{red} & \text{with probability } 1 - p. \end{cases}$$

- How to choose p? Maximise the number of matched vertices.

Colouring vertices

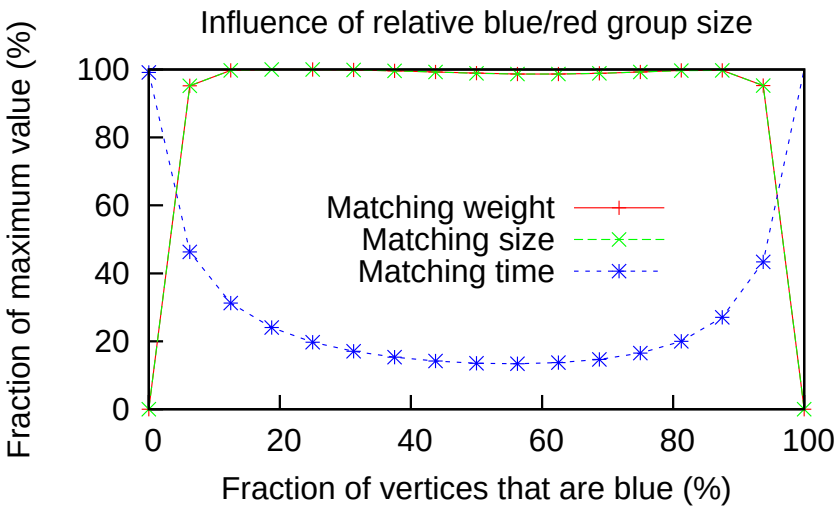
- To colour vertices $v \in V$, we use for a fixed $p \in [0, 1]$

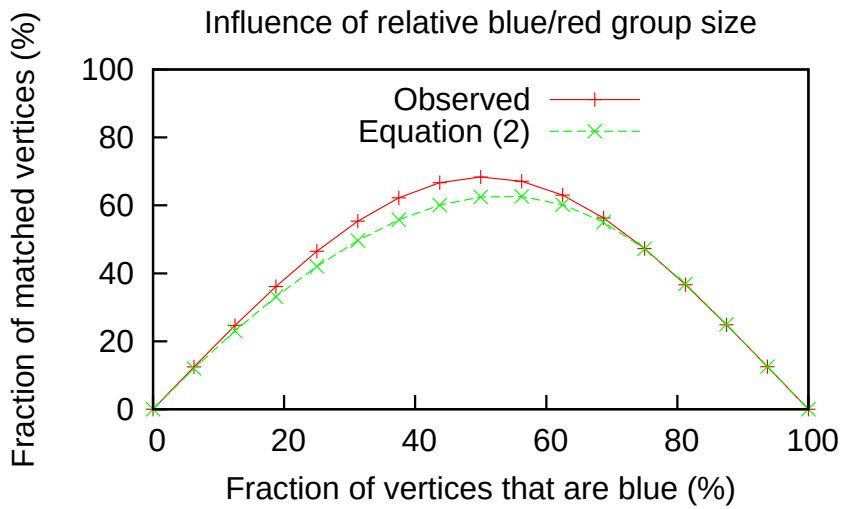
$$\pi(v) = \begin{cases} \textbf{blue} & \text{with probability } p, \\ \textbf{red} & \text{with probability } 1 - p. \end{cases}$$

- How to choose p? Maximise the number of matched vertices.
- For a large random graphs, the expected fraction of matched vertices can be approximated by (independent of edge density)

$$2\,(1-p)\left(1 - e^{-\frac{p}{1-p}}\right).$$

Choosing p





We should choose $p \approx 0.53406$.

Choosing p

- We should maximise the relative number of matched vertices each round.

Choosing p

- We should maximise the relative number of matched vertices each round.
- The number of matched vertices equals twice the number of **red** vertices that receive at least one proposal: maximise $\frac{2N}{|V|}$, where

  $N :=$ number of **red** vertices receiving at least one proposal.

Choosing p

- We should maximise the relative number of matched vertices each round.
- The number of matched vertices equals twice the number of **red** vertices that receive at least one proposal: maximise $\frac{2N}{|V|}$, where

$$N := \text{number of } \textbf{red} \text{ vertices receiving at least one proposal.}$$

- For a random graph with n vertices, we can approximate (independent of edge density)

$$\lim_{n\to\infty} \frac{2\,E(N(n))}{n} \approx 2\,(1-p)\left(1 - e^{-\frac{p}{1-p}}\right).$$

Choosing p

Let $G = (\{1, \ldots, n\}, E)$ with $P(\{v, w\} \in E) = d$ for $d \in]0, 1]$. Then $E(N(n))$ is given by

Choosing p

Let $G = (\{1, \ldots, n\}, E)$ with $P(\{v, w\} \in E) = d$ for $d \in]0, 1]$. Then $E(N(n))$ is given by

$$\sum_{v \in V} P(\pi(v) = \textbf{red})\, P(v \text{ is proposed to} \mid \pi(v) = \textbf{red})$$

Choosing p

Let $G = (\{1, \ldots, n\}, E)$ with $P(\{v, w\} \in E) = d$ for $d \in]0, 1]$. Then $E(N(n))$ is given by

$$\sum_{v \in V} P(\pi(v) = \textbf{red})\, P(v \text{ is proposed to} \mid \pi(v) = \textbf{red})$$

$$= \sum_{v \in V} P(\pi(v) = \textbf{red}) \left(1 - \prod_{w \in V \setminus \{v\}} (1 - P(w \text{ proposes to } v \mid \pi(v) = \textbf{red}))\right)$$

Choosing p

Let $G = (\{1, \ldots, n\}, E)$ with $P(\{v, w\} \in E) = d$ for $d \in]0, 1]$. Then $E(N(n))$ is given by

$$\sum_{v \in V} P(\pi(v) = \textbf{red})\, P(v \text{ is proposed to} \mid \pi(v) = \textbf{red})$$

$$= \sum_{v \in V} P(\pi(v) = \textbf{red}) \left(1 - \prod_{w \in V \setminus \{v\}} (1 - P(w \text{ proposes to } v \mid \pi(v) = \textbf{red}))\right)$$

$$= \sum_{v \in V} P(\pi(v) = \textbf{red}) \left(1 - \prod_{w \in V \setminus \{v\}} \left(1 - \frac{P(\pi(w) = \textbf{blue})\, P(\{v, w\} \in E)}{\text{nr. of } \textbf{red} \text{ neighb. of } w}\right)\right)$$

Choosing p

Let $G = (\{1, \ldots, n\}, E)$ with $P(\{v, w\} \in E) = d$ for $d \in]0, 1]$. Then $E(N(n))$ is given by

$$\sum_{v \in V} P(\pi(v) = \textbf{red})\, P(v \text{ is proposed to} \mid \pi(v) = \textbf{red})$$

$$= \sum_{v \in V} P(\pi(v) = \textbf{red}) \left(1 - \prod_{w \in V \setminus \{v\}} (1 - P(w \text{ proposes to } v \mid \pi(v) = \textbf{red}))\right)$$

$$= \sum_{v \in V} P(\pi(v) = \textbf{red}) \left(1 - \prod_{w \in V \setminus \{v\}} \left(1 - \frac{P(\pi(w) = \textbf{blue})\, P(\{v, w\} \in E)}{\text{nr. of } \textbf{red} \text{ neighb. of } w}\right)\right)$$

$$\approx n(1-p) \left(1 - \left(1 - \frac{p\, d}{1 + (1-p)\,(d\,(n-1) - 1)}\right)^{n-1}\right).$$

GPU coarsening

- **Input:** a graph $G = (V, E, \omega, \zeta)$ and a map $\mu : V \to \mathbb{N}$.

GPU coarsening

- **Input:** a graph $G = (V, E, \omega, \zeta)$ and a map $\mu : V \to \mathbb{N}$.
- **Output:** a graph $G' = (V', E', \omega', \zeta')$ and surjective map $\pi : V \to V'$ such that:

GPU coarsening

- **Input:** a graph $G = (V, E, \omega, \zeta)$ and a map $\mu : V \to \mathbb{N}$.
- **Output:** a graph $G' = (V', E', \omega', \zeta')$ and surjective map $\pi : V \to V'$ such that:
  - $E' = \{\{\pi(u), \pi(v)\} \mid \{u, v\} \in E\}$ (collapse edges),

GPU coarsening

- **Input:** a graph $G = (V, E, \omega, \zeta)$ and a map $\mu : V \to \mathbb{N}$.
- **Output:** a graph $G' = (V', E', \omega', \zeta')$ and surjective map $\pi : V \to V'$ such that:
  - $E' = \{\{\pi(u), \pi(v)\} \mid \{u, v\} \in E\}$ (collapse edges),
  - $$\omega'(e') = \sum_{\pi(e)=e'} \omega(e) \quad \text{(sum edge weights)},$$

GPU coarsening

- **Input:** a graph $G = (V, E, \omega, \zeta)$ and a map $\mu : V \to \mathbb{N}$.
- **Output:** a graph $G' = (V', E', \omega', \zeta')$ and surjective map $\pi : V \to V'$ such that:
  - $E' = \{\{\pi(u), \pi(v)\} \mid \{u, v\} \in E\}$ (collapse edges),
  - $$\omega'(e') = \sum_{\pi(e)=e'} \omega(e) \quad \text{(sum edge weights)},$$
  - $$\zeta'(v') = \sum_{\pi(v)=v'} \zeta(v) \quad \text{(sum vertex weights)},$$

GPU coarsening

- **Input:** a graph $G = (V, E, \omega, \zeta)$ and a map $\mu : V \to \mathbb{N}$.
- **Output:** a graph $G' = (V', E', \omega', \zeta')$ and surjective map $\pi : V \to V'$ such that:
  - $E' = \{\{\pi(u), \pi(v)\} \mid \{u, v\} \in E\}$ (collapse edges),
  - $$\omega'(e') = \sum_{\pi(e)=e'} \omega(e)$$ (sum edge weights),
  - $$\zeta'(v') = \sum_{\pi(v)=v'} \zeta(v)$$ (sum vertex weights),
  - $\pi(u) = \pi(v)$ if and only if $\mu(u) = \mu(v)$ (compress μ to π).

GPU coarsening (implementation)

- Implemented using the CUDA Thrust library.

GPU coarsening (implementation)

- Implemented using the CUDA Thrust library.
- View G as a collection of adjacency lists for each vertex.

GPU coarsening (implementation)

- Implemented using the CUDA Thrust library.
- View G as a collection of adjacency lists for each vertex.
- First, we create π and π^{-1} from μ.

GPU coarsening (implementation)

- Implemented using the CUDA Thrust library.
- View G as a collection of adjacency lists for each vertex.
- First, we create π and π^{-1} from μ.
- Then, we create the new adjacency lists and weights for G'.

GPU coarsening (implementation)

- Implemented using the CUDA Thrust library.
- View G as a collection of adjacency lists for each vertex.
- First, we create π and π^{-1} from μ.
- Then, we create the new adjacency lists and weights for G'.
- Use μ, π, π^{-1}, and a bookkeeping array ρ in global GPU memory.

GPU coarsening (algorithm)

ρ	1	2	3	4	5	6	7	8	9	10	11	12
μ	9	2	3	22	9	9	22	2	3	3	2	4
π^{-1}												
π												

Initialise ρ sequentially and store μ.

GPU coarsening (algorithm)

ρ	1	2	3	4	5	6	7	8	9	10	11	12
μ	9	2	3	22	9	9	22	2	3	3	2	4
π^{-1}												
π												

Sort by increasing μ-value (`sort_by_key`).

GPU coarsening (algorithm)

ρ	2	8	11	3	9	10	12	1	5	6	4	7
μ	2	2	2	3	3	3	4	9	9	9	22	22
π^{-1}												
π												

Sort by increasing μ-value (`sort_by_key`).

GPU coarsening (algorithm)

ρ	2	8	11	3	9	10	12	1	5	6	4	7
μ	2	2	2	3	3	3	4	9	9	9	22	22
π^{-1}												
π												

Determine different matched groups (`adjacent_not_equal`).

GPU coarsening (algorithm)

ρ	2	8	11	3	9	10	12	1	5	6	4	7
μ	1	0	0	1	0	0	1	1	0	0	1	0
π^{-1}												
π												

Determine different matched groups (`adjacent_not_equal`).

GPU coarsening (algorithm)

ρ	2	8	11	3	9	10	12	1	5	6	4	7
μ	1	0	0	1	0	0	1	1	0	0	1	0
π^{-1}												
π												

Extract boundaries for π^{-1} (`copy_index_if_nonzero`).

GPU coarsening (algorithm)

ρ	2	8	11	3	9	10	12	1	5	6	4	7
μ	1	0	0	1	0	0	1	1	0	0	1	0
π^{-1}	1	4	7	8	11	13						
π												

Extract boundaries for π^{-1} (`copy_index_if_nonzero`).

GPU coarsening (algorithm)

ρ	2	8	11	3	9	10	12	1	5	6	4	7
μ	1	0	0	1	0	0	1	1	0	0	1	0
π^{-1}	1	4	7	8	11	13						
π												

Perform scan to find π indices (`inclusive_scan`).

GPU coarsening (algorithm)

ρ	2	8	11	3	9	10	12	1	5	6	4	7
μ	1	1	1	2	2	2	3	4	4	4	5	5
π^{-1}	1	4	7	8	11	13						
π												

Perform scan to find π indices (`inclusive_scan`).

GPU coarsening (algorithm)

ρ	2	8	11	3	9	10	12	1	5	6	4	7
μ	1	1	1	2	2	2	3	4	4	4	5	5
π^{-1}	1	4	7	8	11	13						
π												

Extract π as $\pi(\rho(i)) = \mu(i)$ (`scatter`).

GPU coarsening (algorithm)

ρ	2	8	11	3	9	10	12	1	5	6	4	7
μ	1	1	1	2	2	2	3	4	4	4	5	5
π^{-1}	1	4	7	8	11	13						
π	4	1	2	5	4	4	5	1	2	2	1	3

Extract π as $\pi(\rho(i)) = \mu(i)$ (`scatter`).

GPU coarsening (algorithm)

- Construct the adjacency lists of G' for $i' \in V'$ in parallel.

GPU coarsening (algorithm)

- Construct the adjacency lists of G' for $i' \in V'$ in parallel.
- Sum vertex weights:

$$\zeta'(i') = \zeta(\rho(\pi^{-1}(i'))) + \ldots + \zeta(\rho(\pi^{-1}(i'+1)-1)).$$

GPU coarsening (algorithm)

- Construct the adjacency lists of G' for $i' \in V'$ in parallel.
- Sum vertex weights:

$$\zeta'(i') = \zeta(\rho(\pi^{-1}(i'))) + \ldots + \zeta(\rho(\pi^{-1}(i'+1)-1)).$$

- Gather weighted neighbours of $\rho(\pi^{-1}(i'))$ to $\rho(\pi^{-1}(i'+1)-1)$:

$$(i_1, \omega_1, i_2, \omega_2, \ldots, i_k, \omega_k).$$

GPU coarsening (algorithm)

- Construct the adjacency lists of G' for $i' \in V'$ in parallel.
- Sum vertex weights:

$$\zeta'(i') = \zeta(\rho(\pi^{-1}(i'))) + \ldots + \zeta(\rho(\pi^{-1}(i'+1)-1)).$$

- Gather weighted neighbours of $\rho(\pi^{-1}(i'))$ to $\rho(\pi^{-1}(i'+1)-1)$:

$$(\pi(i_1), \omega_1, \pi(i_2), \omega_2, \ldots, \pi(i_k), \omega_k).$$

GPU coarsening (algorithm)

- Construct the adjacency lists of G' for $i' \in V'$ in parallel.
- Sum vertex weights:

$$\zeta'(i') = \zeta(\rho(\pi^{-1}(i'))) + \ldots + \zeta(\rho(\pi^{-1}(i'+1)-1)).$$

- Gather weighted neighbours of $\rho(\pi^{-1}(i'))$ to $\rho(\pi^{-1}(i'+1)-1)$:

$$(\pi(i_1), \omega_1, \pi(i_2), \omega_2, \ldots, \pi(i_k), \omega_k).$$

- Sort the neighbour list by index.

GPU coarsening (algorithm)

- Construct the adjacency lists of G' for $i' \in V'$ in parallel.
- Sum vertex weights:

$$\zeta'(i') = \zeta(\rho(\pi^{-1}(i'))) + \ldots + \zeta(\rho(\pi^{-1}(i'+1)-1)).$$

- Gather weighted neighbours of $\rho(\pi^{-1}(i'))$ to $\rho(\pi^{-1}(i'+1)-1)$:

$$(\pi(i_1), \omega_1, \pi(i_2), \omega_2, \ldots, \pi(i_k), \omega_k).$$

- Sort the neighbour list by index.
- Compress the neighbour list by replacing subsequences $(j', \omega_1, j', \omega_2, \ldots, j', \omega_l)$ with $(j', \omega_1 + \omega_2 + \ldots + \omega_l)$.