Computational Tools for Data Science

Week 7:

Mining social network graphs

Recap: Graphs

- A (**simple** and **undirected**) **graph** is a pair $\boldsymbol{G} = (\boldsymbol{V}(\boldsymbol{G}), \boldsymbol{E}(\boldsymbol{G}))$.

- The set $\boldsymbol{V}(\boldsymbol{G})$ is the **vertex set** of the graph G. Its elements are the **vertices** of G (sometimes they are also called **nodes**).

Recap: Graphs

- A (**simple** and **undirected**) **graph** is a pair $\boldsymbol{G} = (\boldsymbol{V}(\boldsymbol{G}), \boldsymbol{E}(\boldsymbol{G}))$.

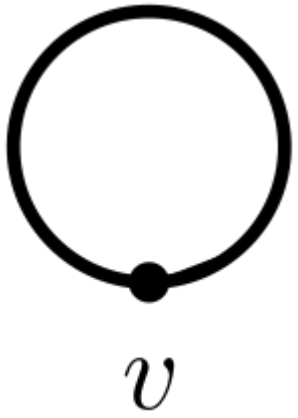
- The set $\boldsymbol{V}(\boldsymbol{G})$ is the **vertex set** of the graph G. Its elements are the **vertices** of G (sometimes they are also called **nodes**).

- The set $\boldsymbol{E}(\boldsymbol{G})$ is the **edge set** of G. Its elements are the **edges** of G.
  - An edge $e \in E(G)$ is a **2-element subset** of the vertex set $V(G)$.
    Hence, $e = \{u, v\}$ for some vertices $u, v \in V(G)$.
    We briefly write uv for an edge $\{u, v\}$.

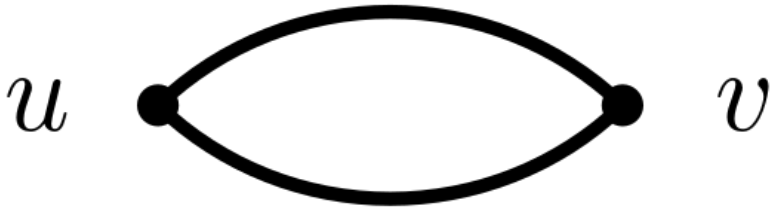
Recap: Graphs

Simple graphs:

1. **No loops:**



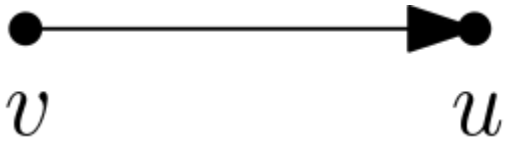
2. **No parallel edges:**



Recap: Graphs

Undirected graphs… what are **directed graphs** (briefly digraphs):

Edges are directed:



Formally, edges are no longer 2-element subsets of $V(G)$.

Model a **directed edge** $\boldsymbol{e}$ (also called **arc**) as a triple $(\boldsymbol{e}, \boldsymbol{v}, \boldsymbol{u})$, meaning that the arc e is directed from vertex v to vertex u.

Graphs as models for networks

Examples:

- Transportation network
- Electric circuits
- Many types of flows: traffic flow, electric flow …
- Phylogenetic networks (more complex than phylogenetic trees)
- Timetables and assignments with priorities (Nobel prize: Shapley)
- Internet

Social networks

Examples:

- Communication networks: telephone networks, email networks
- Financial transactions
- Collaboration networks
- Social media, e.g.:
  - Facebook → **friends relation** corresponds to **undirected edge**
  - Twitter → **follow relation** corresponds to **directed edge**
- (Internet)

Modelling social networks via graphs

1. Vertices correspond to network participants:
   people, addresses, websites, accounts

Modelling social networks via graphs

1. Vertices correspond to network participants:

   people, addresses, websites, accounts

2. Edges correspond to relationship between two participants:

   friends or no friends (undirected edge)

   following (directed edge)

   flow of transactions (weighted edge)

   different types of relationship (coloured edge)

Modelling social networks via graphs

1. Vertices correspond to network participants:
   people, addresses, websites, accounts

2. Edges correspond to relationship between two participants:
   friends or no friends (undirected edge)
   following (directed edge)
   flow of transactions (weighted edge)
   different types of relationship (coloured edge)

3. **Locality property**

Modelling social networks via graphs

3. Locality property

We assume the network is not random.

Relations within the network tend to cluster in communities.

Modelling social networks via graphs

3. Locality property

We assume the network is not random.

Relations within the network tend to cluster in communities.

If A is related to B and B is related to C, then the probability of A and C being related is higher than the average.

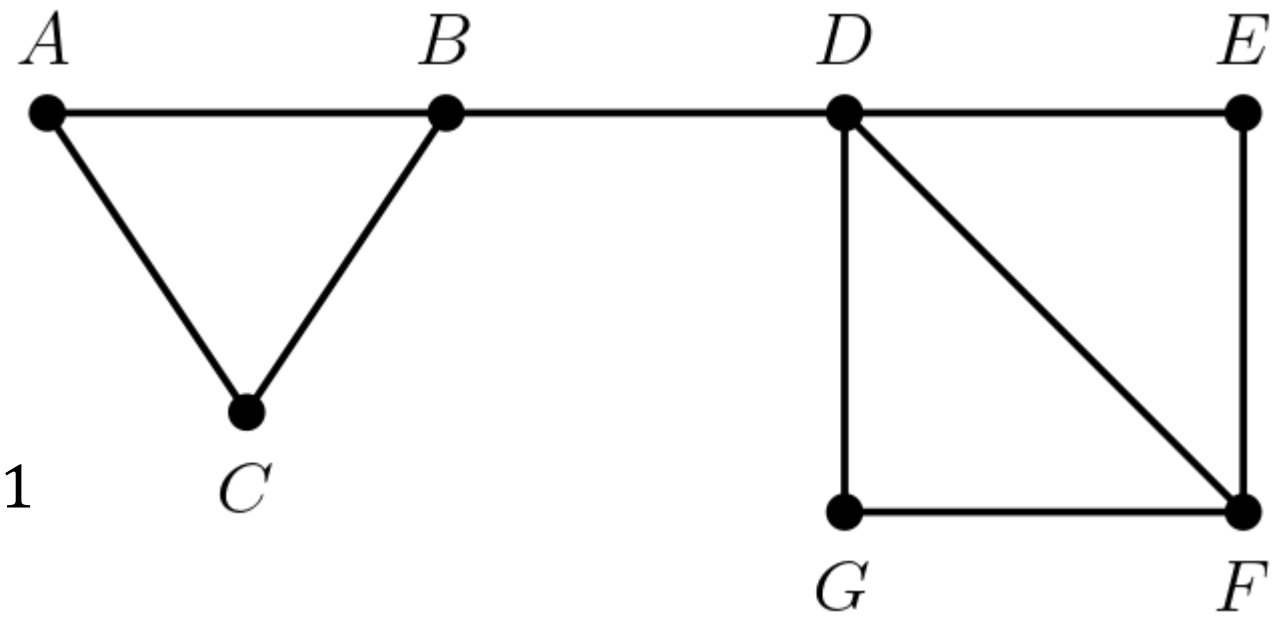
Modelling social networks via graphs

Example

$$|E(G)| = 9$$

$$|V(G)| = 7$$

$$\left|\binom{V(G)}{2}\right| = \binom{7}{2} = 21$$



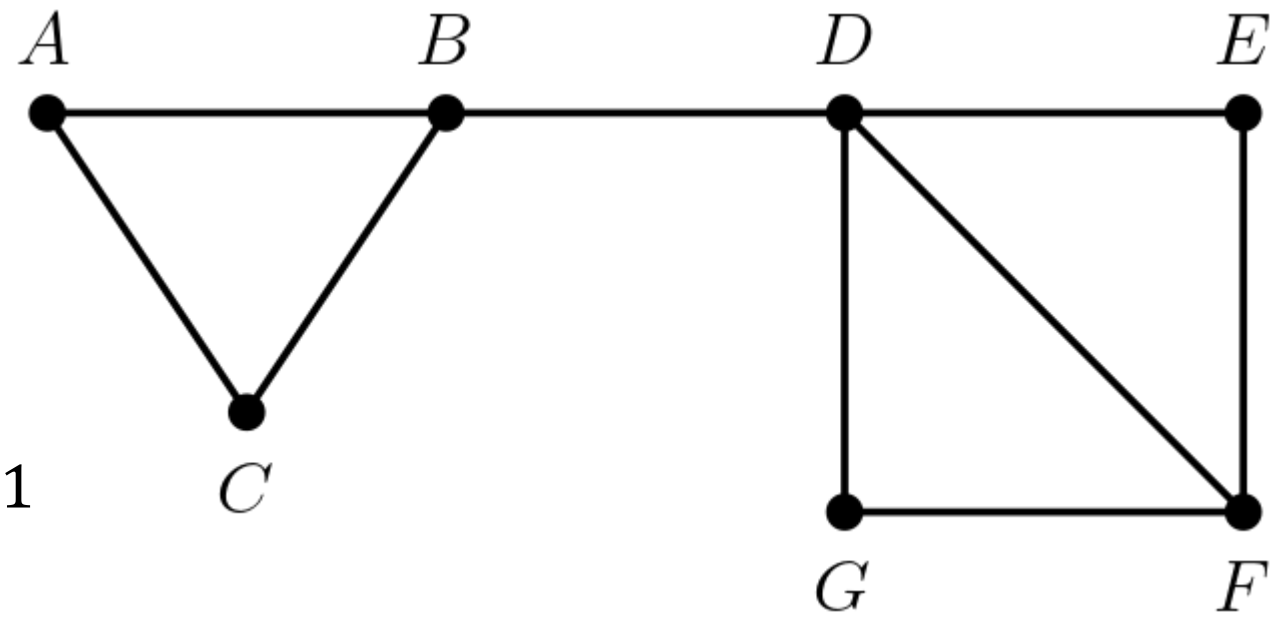
Modelling social networks via graphs

Example

$|E(G)| = 9$

$|V(G)| = 7$

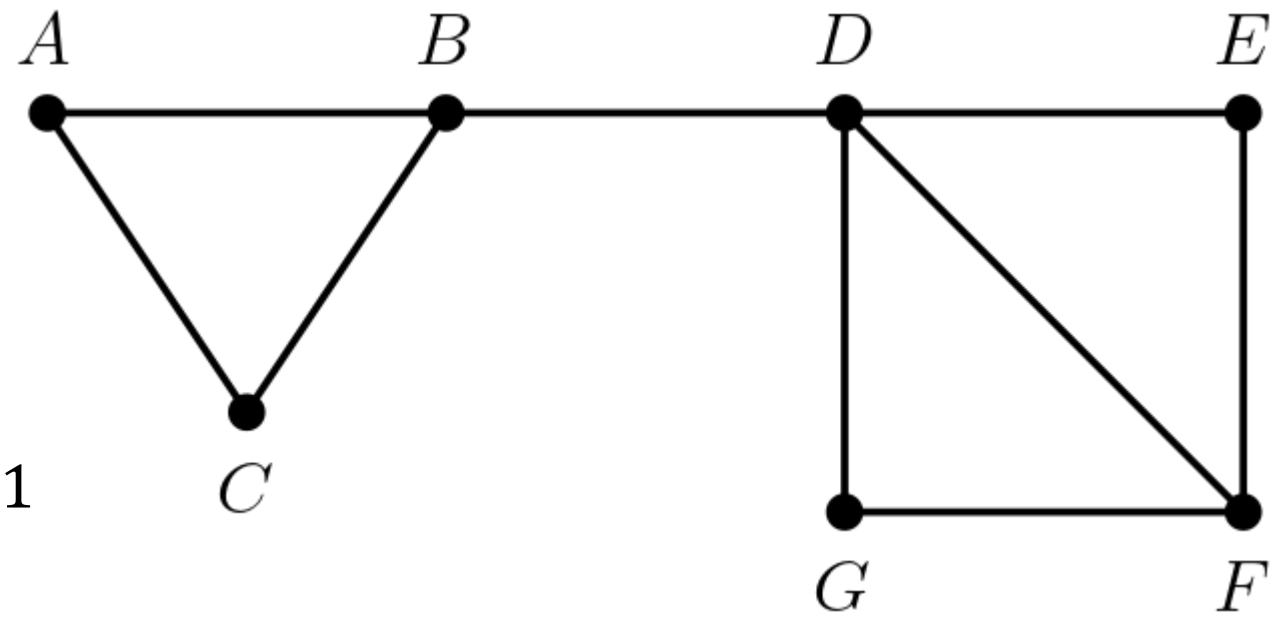
$$\left|\binom{V(G)}{2}\right| = \binom{7}{2} = 21$$



Suppose $\{X, Y\}, \{Y, Z\} \in E(G)$. Check $\{X, Z\}$:

Modelling social networks via graphs

Example



$|E(G)| = 9$

$|V(G)| = 7$

$$\left|\binom{V(G)}{2}\right| = \binom{7}{2} = 21$$

Suppose $\{X, Y\}, \{Y, Z\} \in E(G)$. Check $\{X, Z\}$:

Random: probability for $\{X, Z\} \in E(G)$: $\frac{|E(G)|-2}{\left|\binom{V(G)}{2}\right|-2} = \frac{7}{19} \approx 0{,}368$

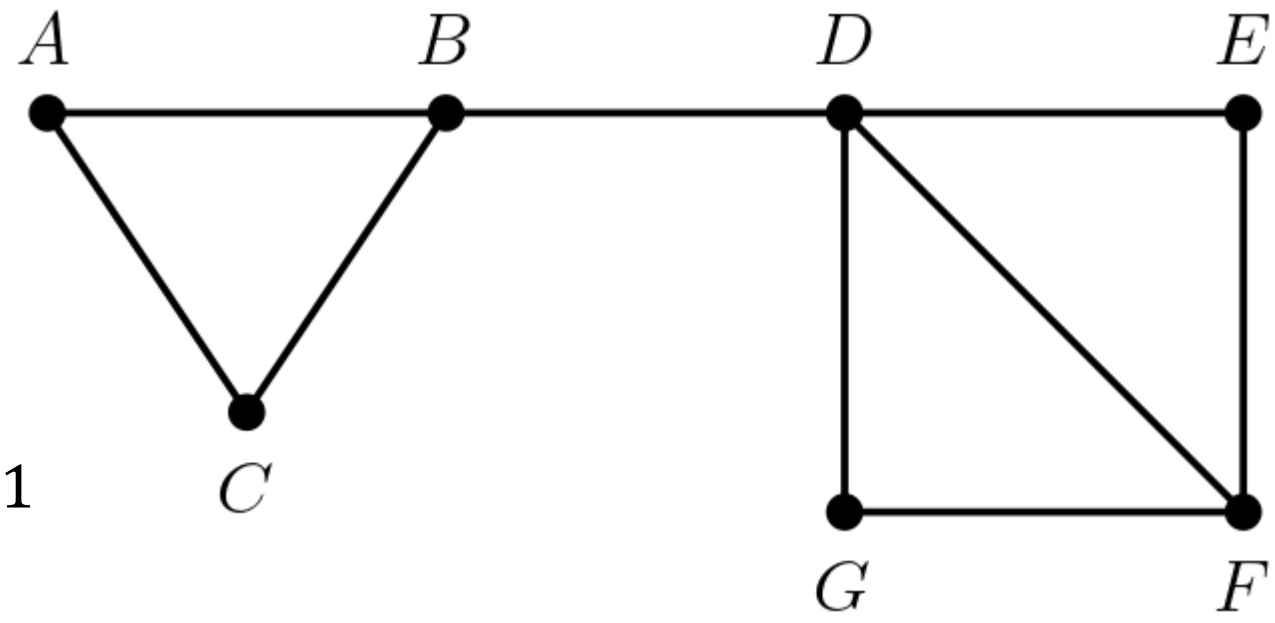
Modelling social networks via graphs

Example

$|E(G)| = 9$

$|V(G)| = 7$

$$\left|\binom{V(G)}{2}\right| = \binom{7}{2} = 21$$

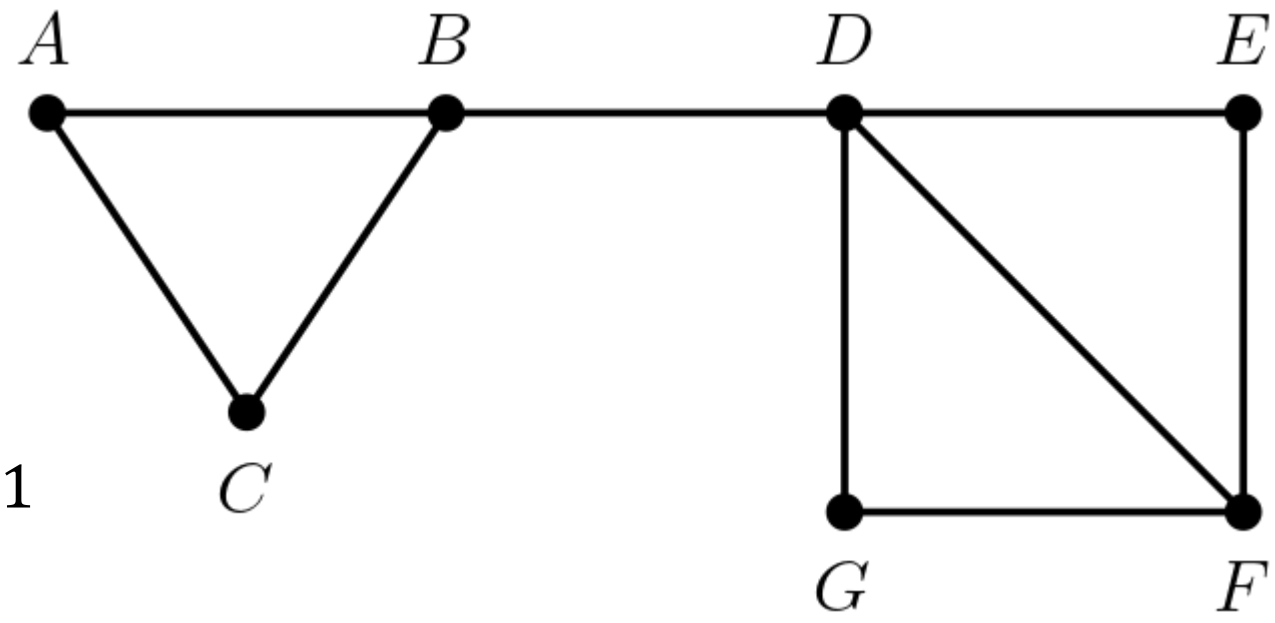


Suppose $\{X, Y\}, \{Y, Z\} \in E(G)$. Check $\{X, Z\}$:

Actually: If $Y = A$, then $\{X, Z\} = \{B, C\}$. Fits locality.

Modelling social networks via graphs

Example



$|E(G)| = 9$

$|V(G)| = 7$

$$\left|\binom{V(G)}{2}\right| = \binom{7}{2} = 21$$

Suppose $\{X, Y\}, \{Y, Z\} \in E(G)$. Check $\{X, Z\}$:

Actually: Same for $Y = C, G, E$

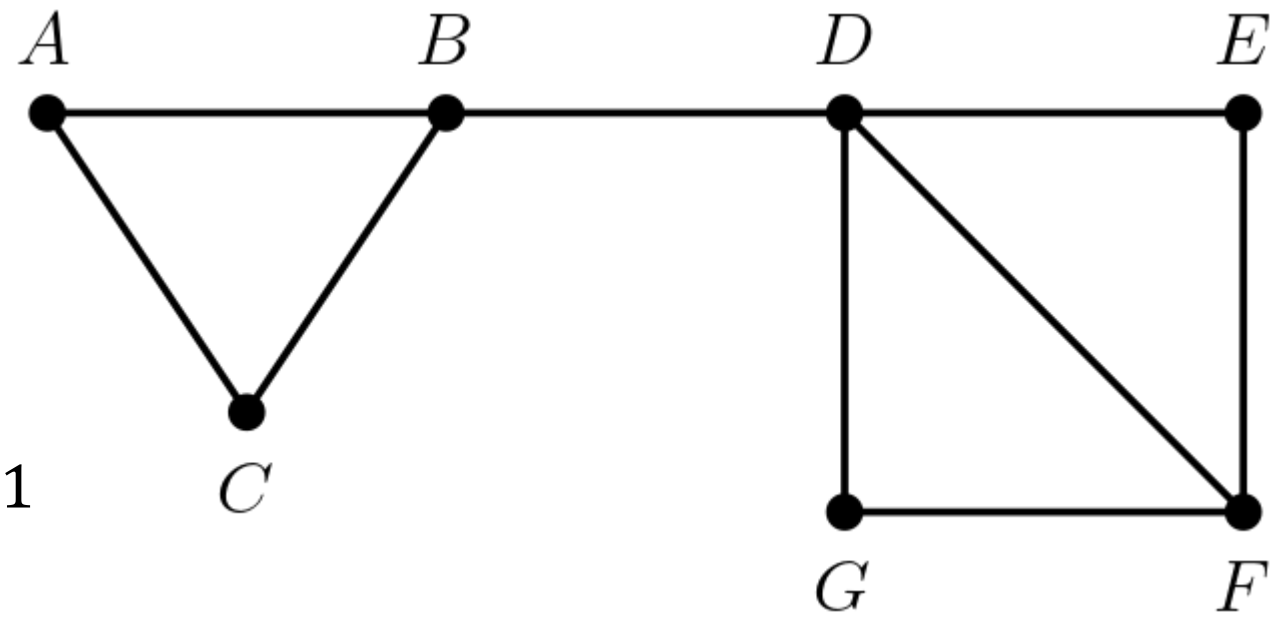
Modelling social networks via graphs

Example

$|E(G)| = 9$

$|V(G)| = 7$

$$\left|\binom{V(G)}{2}\right| = \binom{7}{2} = 21$$



Suppose $\{X, Y\}, \{Y, Z\} \in E(G)$. Check $\{X, Z\}$:

Actually: If $Y = F$, then $\{X, Z\} = \{D, G\}$ and $\{X, Z\} = \{D, E\}$ fit locality.

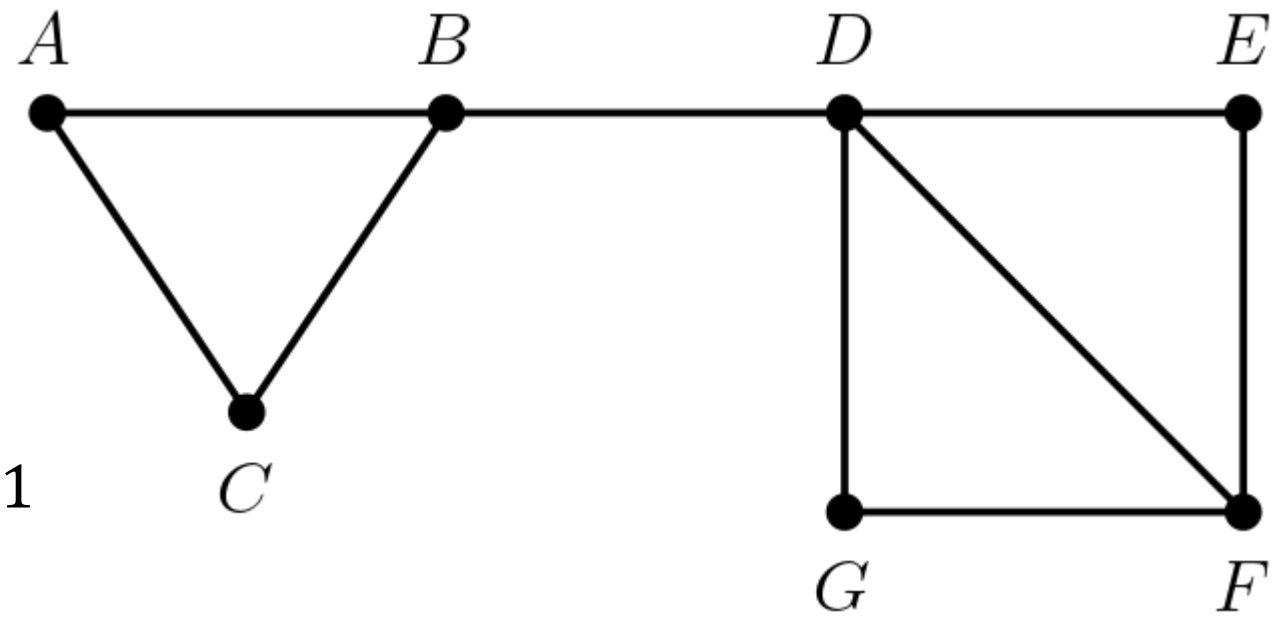
Modelling social networks via graphs

Example

$|E(G)| = 9$

$|V(G)| = 7$

$$\left|\binom{V(G)}{2}\right| = \binom{7}{2} = 21$$



Suppose $\{X, Y\}, \{Y, Z\} \in E(G)$. Check $\{X, Z\}$:

Actually: If $Y = F$, $\{X, Z\} = \{E, G\}$ violates locality.

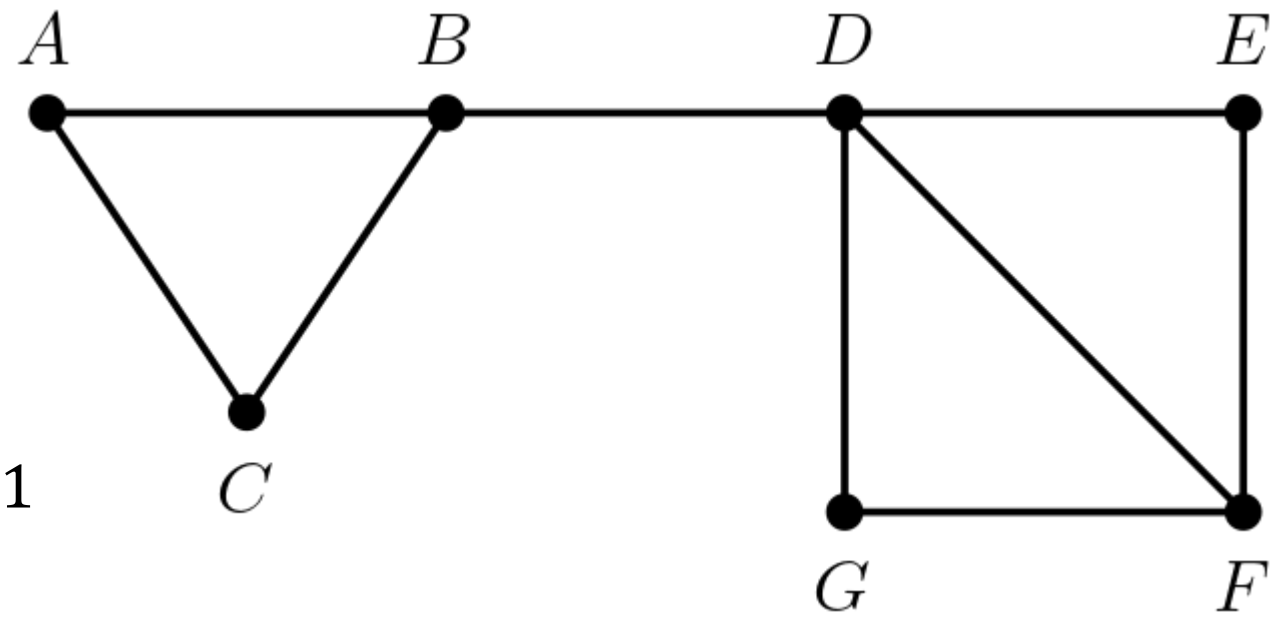
Modelling social networks via graphs

Example

$|E(G)| = 9$

$|V(G)| = 7$

$$\left|\binom{V(G)}{2}\right| = \binom{7}{2} = 21$$

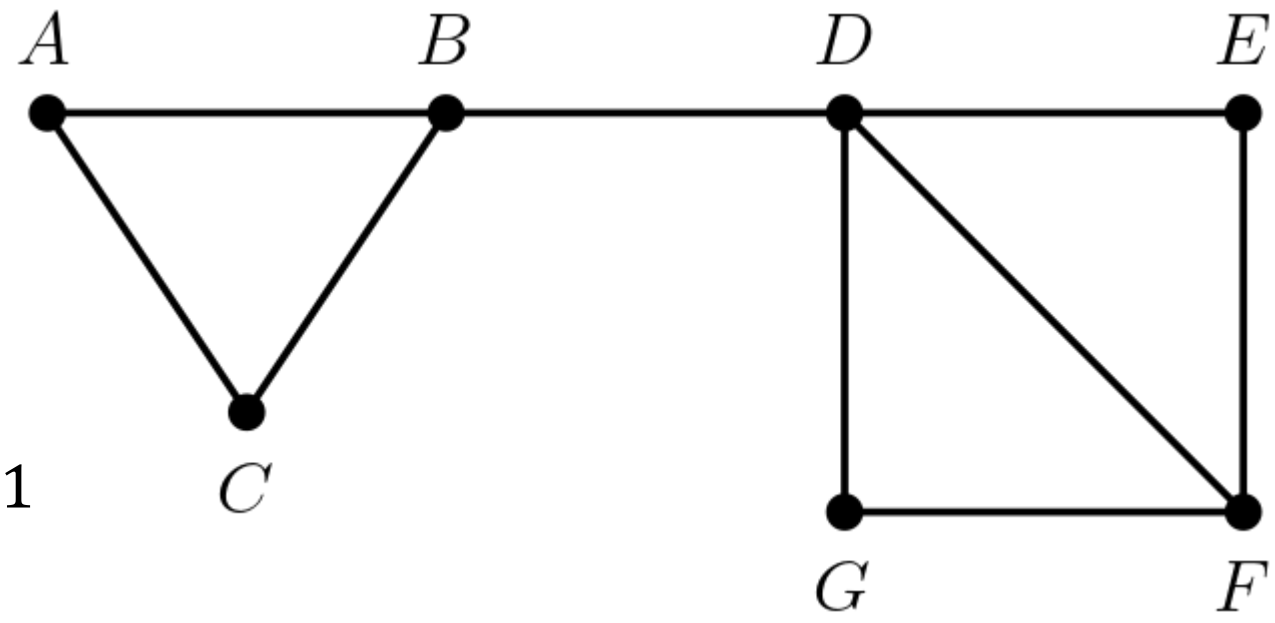


Suppose $\{X, Y\}, \{Y, Z\} \in E(G)$. Check $\{X, Z\}$:

Actually: If $Y = F$, $\{X, Z\} = \{E, G\}$ violates locality. **So far: 6 vs. 1**.

Modelling social networks via graphs

Example



$|E(G)| = 9$

$|V(G)| = 7$

$$\left|\binom{V(G)}{2}\right| = \binom{7}{2} = 21$$

Suppose $\{X, Y\}, \{Y, Z\} \in E(G)$. Check $\{X, Z\}$:

Actually: If $Y = B$: 3 neighbours, only 2 of them adjacent. **7 vs. 3**

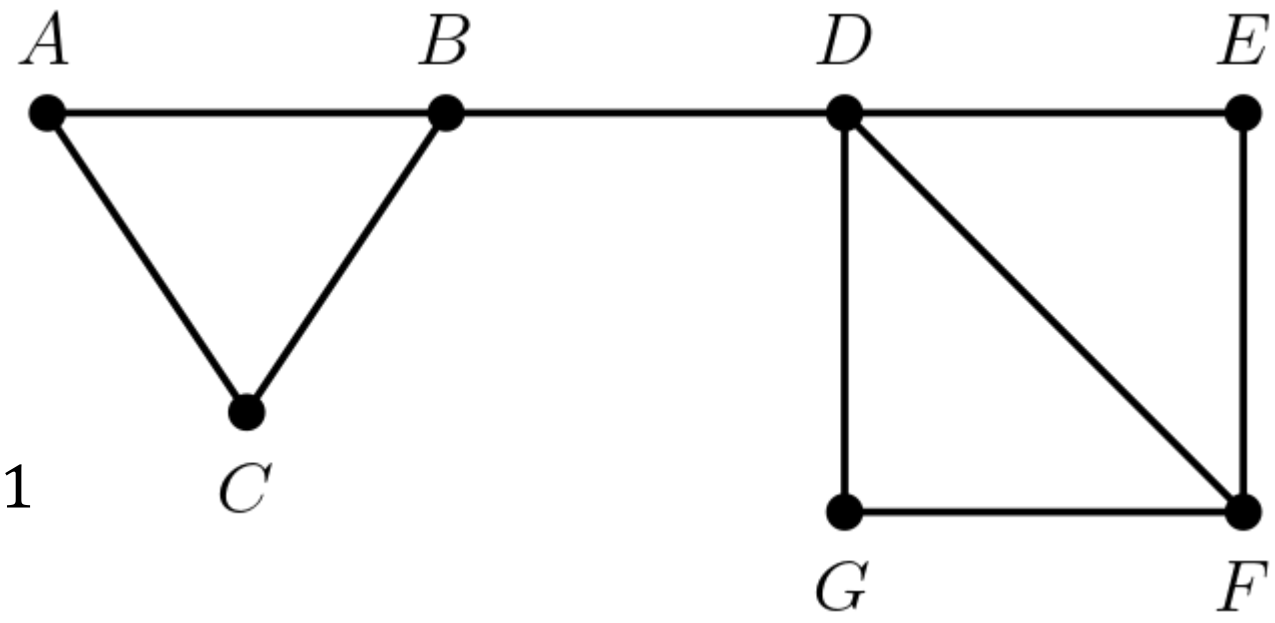
Modelling social networks via graphs

Example

$|E(G)| = 9$

$|V(G)| = 7$

$$\left|\binom{V(G)}{2}\right| = \binom{7}{2} = 21$$



Suppose $\{X, Y\}, \{Y, Z\} \in E(G)$. Check $\{X, Z\}$:

Actually: If $Y = D$: 4 neighbours, 2 pairs of them adjacent. **9 vs. 7**

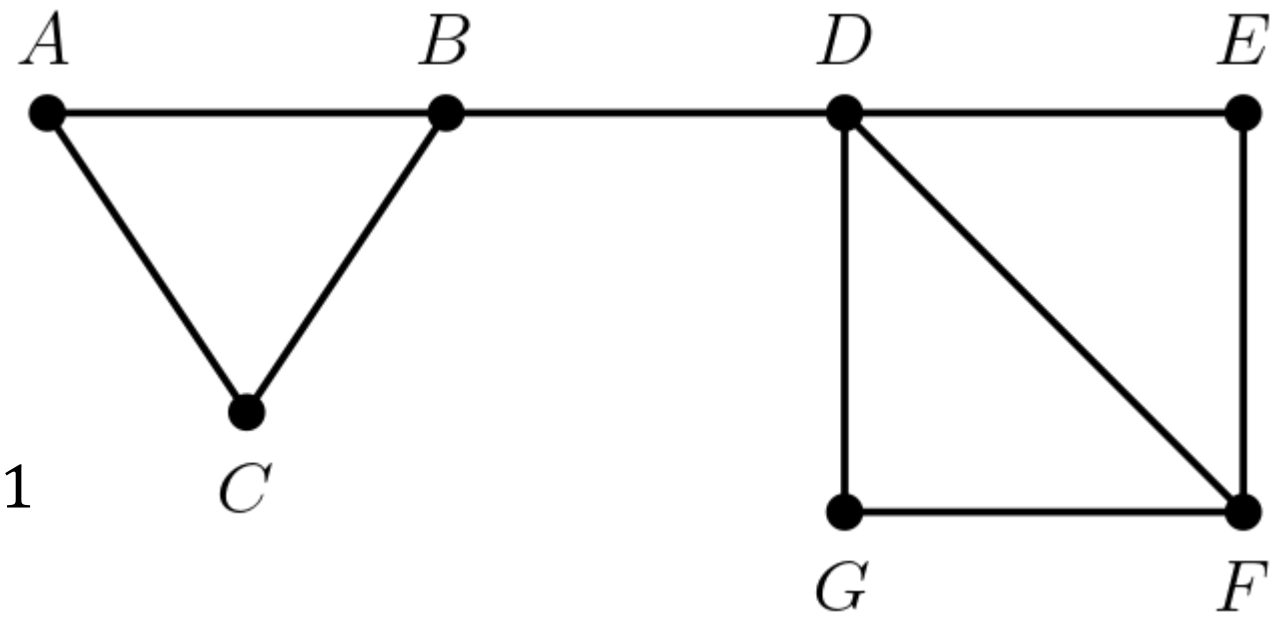
Modelling social networks via graphs

Example

$|E(G)| = 9$

$|V(G)| = 7$

$$\left|\binom{V(G)}{2}\right| = \binom{7}{2} = 21$$



Suppose $\{X, Y\}, \{Y, Z\} \in E(G)$. Check $\{X, Z\}$:

Actually: Fraction the number of times edge $\{X, Z\}$ exists: $\frac{9}{9+7} \approx 0{,}563$

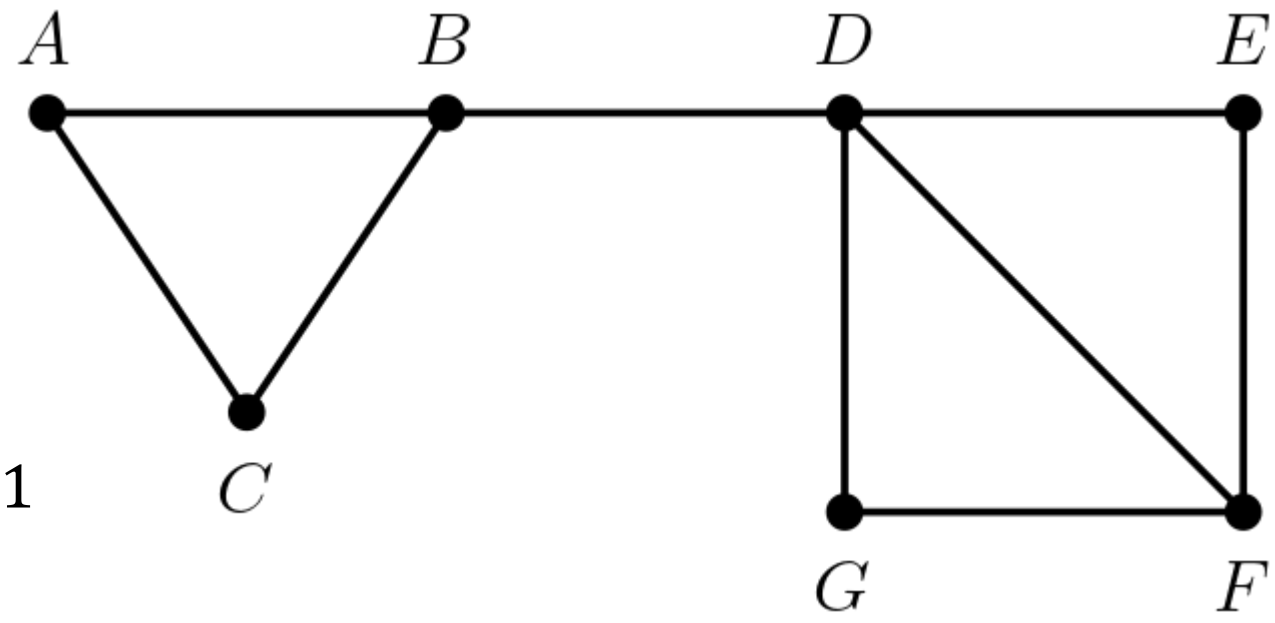
Modelling social networks via graphs

Example

$|E(G)| = 9$

$|V(G)| = 7$

$\left|\binom{V(G)}{2}\right| = \binom{7}{2} = 21$



Conclusion: $0{,}563 \gg 0{,}368$. So locality property holds. The graph might be suitable to model a social network.

Clustering (partitioning) social network graphs

Problems with previously introduced clustering tools.

Examples:

1. Agglomerative hierarchical clustering

2. Point assignment tools (e.g. k-means)

Problems with agglomerative clustering

Let $u, v \in V(G)$. The usual **distance** function d on a graph is:

$$d(u,v) = length\ of\ \boldsymbol{shortest\ path}\ between\ u\ and\ v\ in\ G$$

Problems with agglomerative clustering

Let $u, v \in V(G)$. The usual **distance** function d on a graph is:

$$d(u,v) = length\ of\ \boldsymbol{shortest\ path}\ between\ u\ and\ v\ in\ G$$

Say we measure **distance between two clusters** C_1 and C_2 via shortest distance between two members, one from each cluster.

Problems with agglomerative clustering

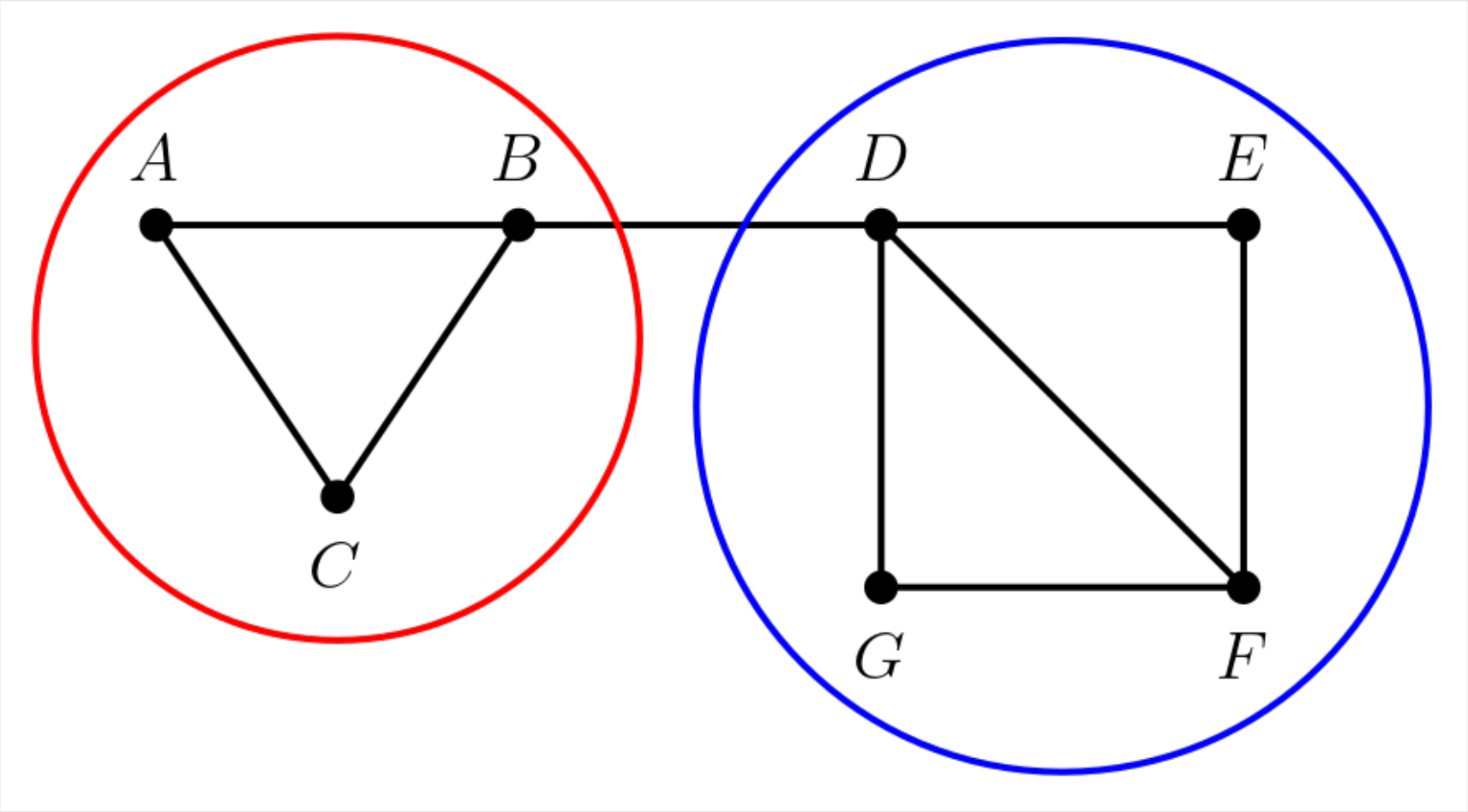
Let $u, v \in V(G)$. The usual **distance** function d on a graph is:

$$d(u,v) = length\ of\ \boldsymbol{shortest\ path}\ between\ u\ and\ v\ in\ G$$

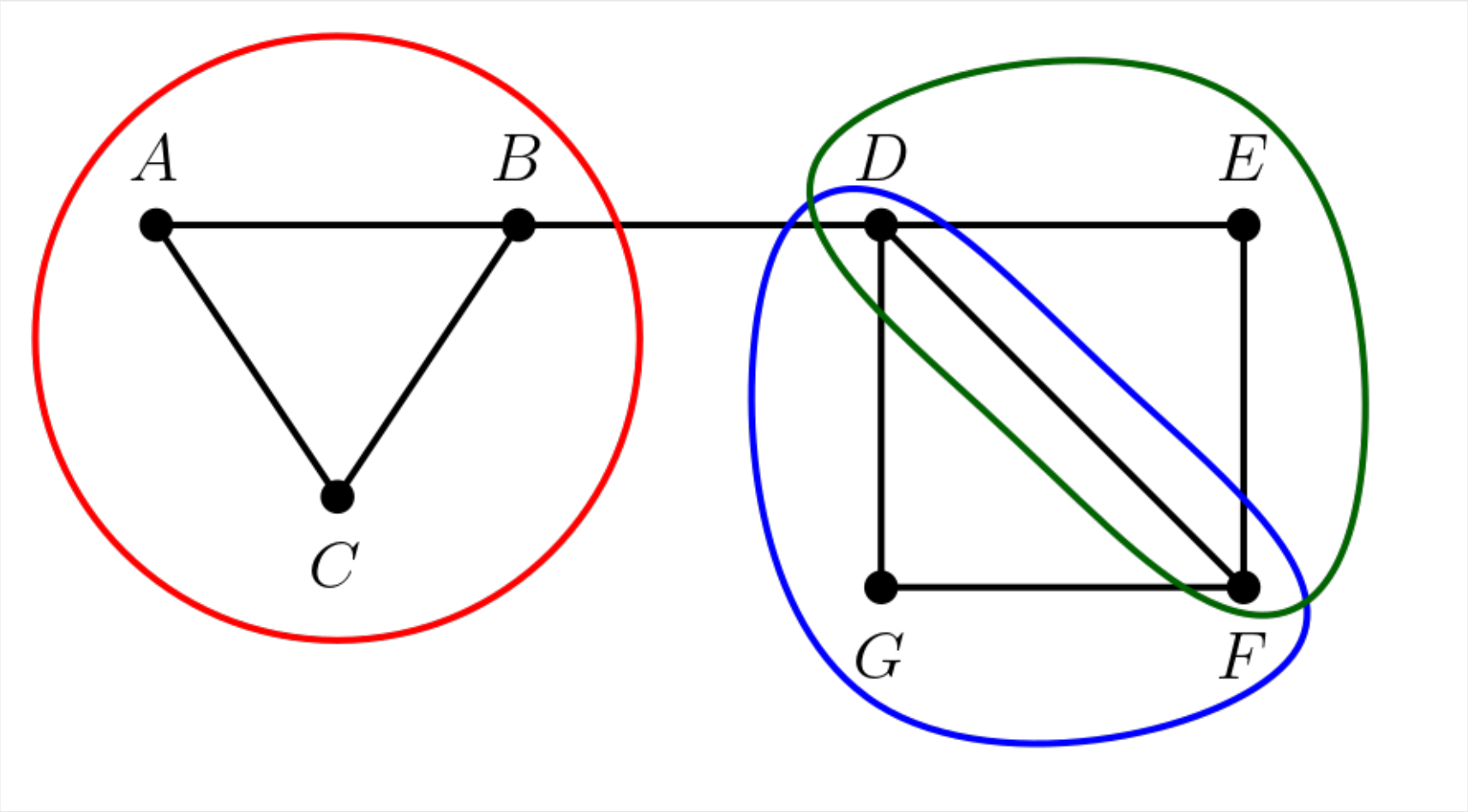
Say we measure **distance between two clusters** C_1 and C_2 via shortest distance between two members, one from each cluster.

Hence, we **always** merge two clusters (or vertices) that are directly connected by an edge (yielding shortest possible distance, namely 1).

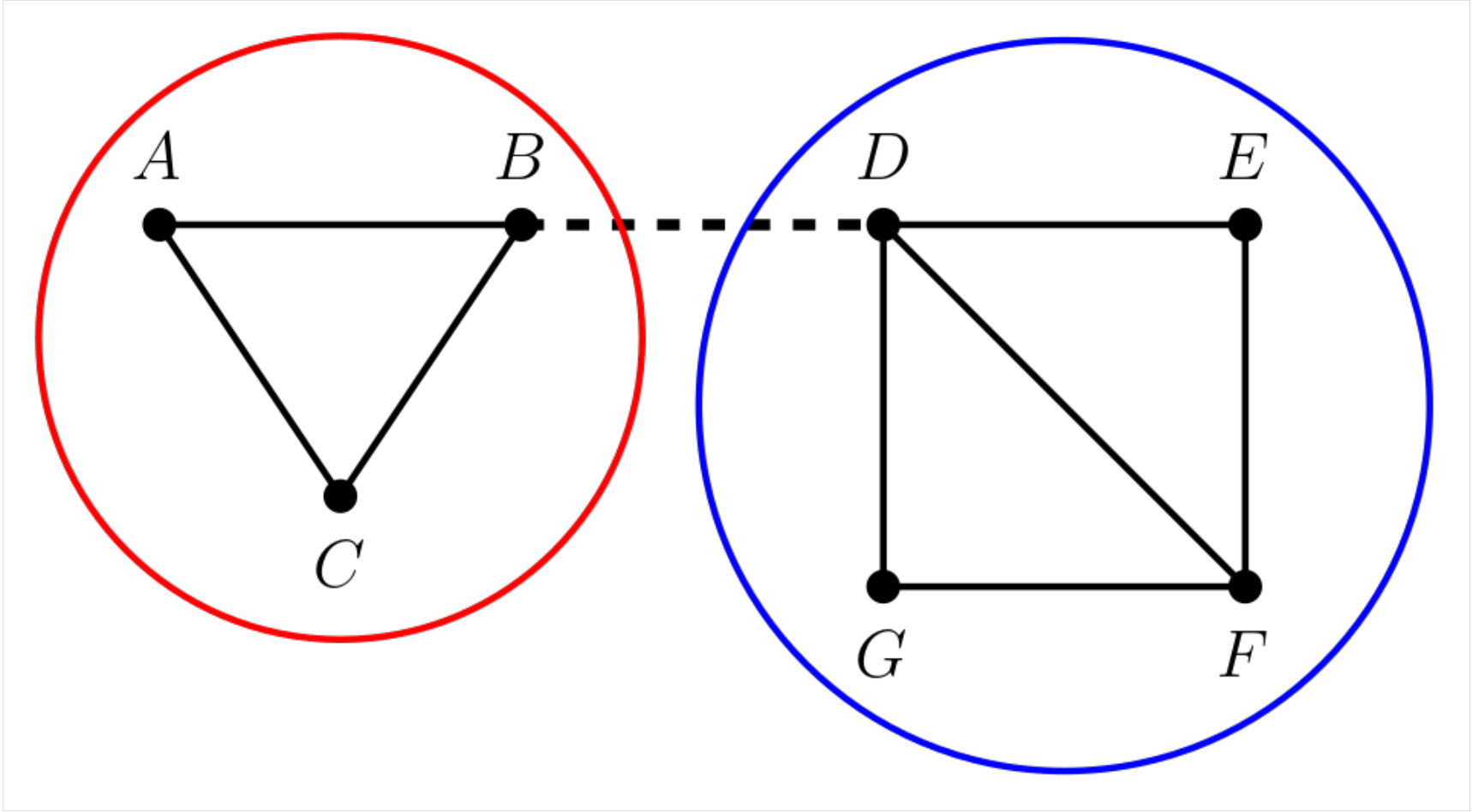
Problems with agglomerative clustering



Problems with agglomerative clustering



Problems with agglomerative clustering



Problems with agglomerative clustering

- Let $e \in E(G)$ whose end vertices lie in different clusters C_1^e, C_2^e. In each merging step, the probability of merging the clusters C_1^e and C_2^e is the same for each such edge e.

- At some point (maybe even initially) it becomes likely that we merge two clusters that should not be combined.

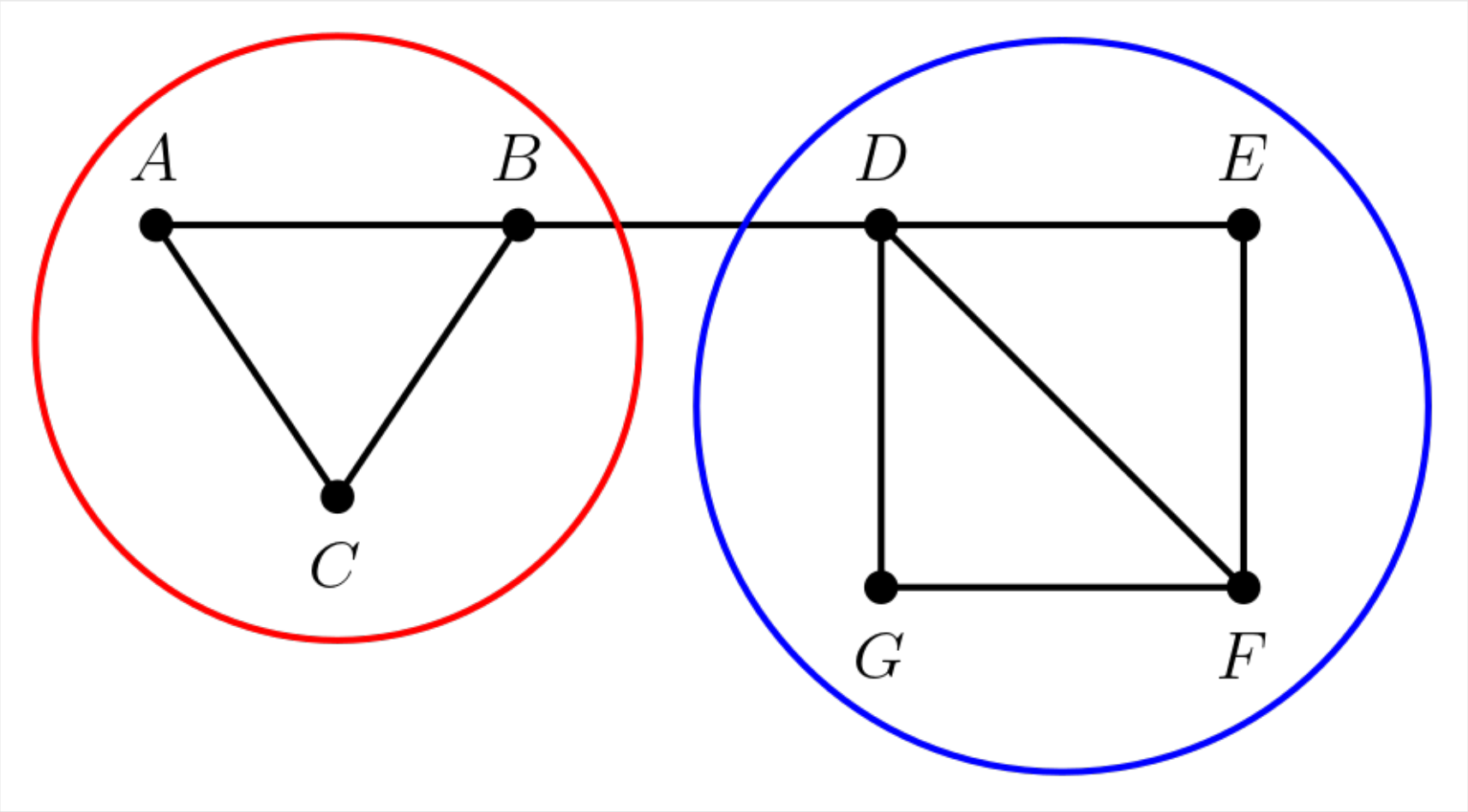
Problems with agglomerative clustering

- Let $e \in E(G)$ whose end vertices lie in different clusters C_1^e, C_2^e. In each merging step, the probability of merging the clusters C_1^e and C_2^e is the same for each such edge e.

- At some point (maybe even initially) it becomes likely that we merge two clusters that should not be combined.

- (We might **prevent this by more sophisticated methods**, e.g. only merge / stop merging when density / cohesion becomes too low, but the **naïve approach is not suitable**.)

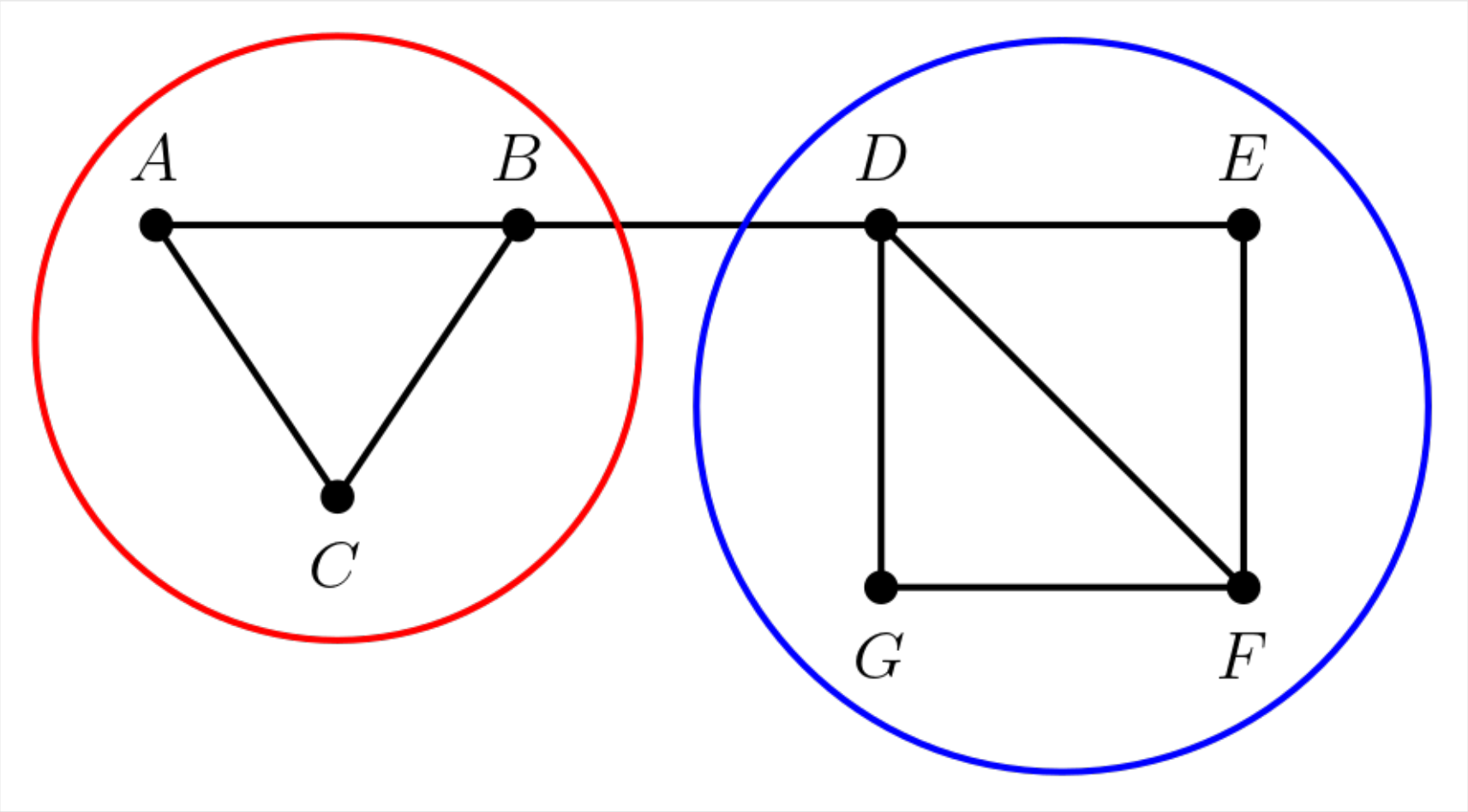
Problems with point assignment (e.g. k-means)



Problems with point assignment (e.g. k-means)

- Say we set $k = 2$ (fitting to our example graph).
- Say our first 2 clustroids are B (at random) and F (far apart from B).

Problems with point assignment (e.g. k-means)



Problems with point assignment (e.g. k-means)

- Say we set $k = 2$ (fitting to our example graph).
- Say our first 2 clustroids are B (at random) and F (far apart from B).
- A and C are assigned to B's cluster.
- E and G are assigned to F's cluster.
- Assigning D to B's or F's cluster is equally reasonable.
  - Hence, with probability 0,5 vertex D ends up in the wrong cluster.

Betweenness

Betweenness centrality

Label each edge $e = \{u, v\} \in E(G)$ with a score $b(e)$.

Betweenness centrality

Label each edge $e = \{u, v\} \in E(G)$ with a score $b(e)$. Define:

$$b(e) = \sum_{\substack{x,y \in V(G) \\ x \neq y}} \frac{\#\ shortest\ x - y\ paths\ that\ use\ e}{\#\ all\ shortest\ x - y\ paths}$$

Betweenness centrality

Label each edge $e = \{u, v\} \in E(G)$ with a score $b(e)$. Define:

$$b(e) = \sum_{\substack{x,y \in V(G) \\ x \neq y}} \frac{\#\ shortest\ x - y\ paths\ that\ use\ e}{\#\ all\ shortest\ x - y\ paths}$$

This labelling is called the **betweenness centrality** (sometimes also just called betweenness) **of the edge e**. The betweenness (centrality) for vertices is defined analogously.

Betweenness centrality

Label each edge $e = \{u, v\} \in E(G)$ with a score $b(e)$. Define:

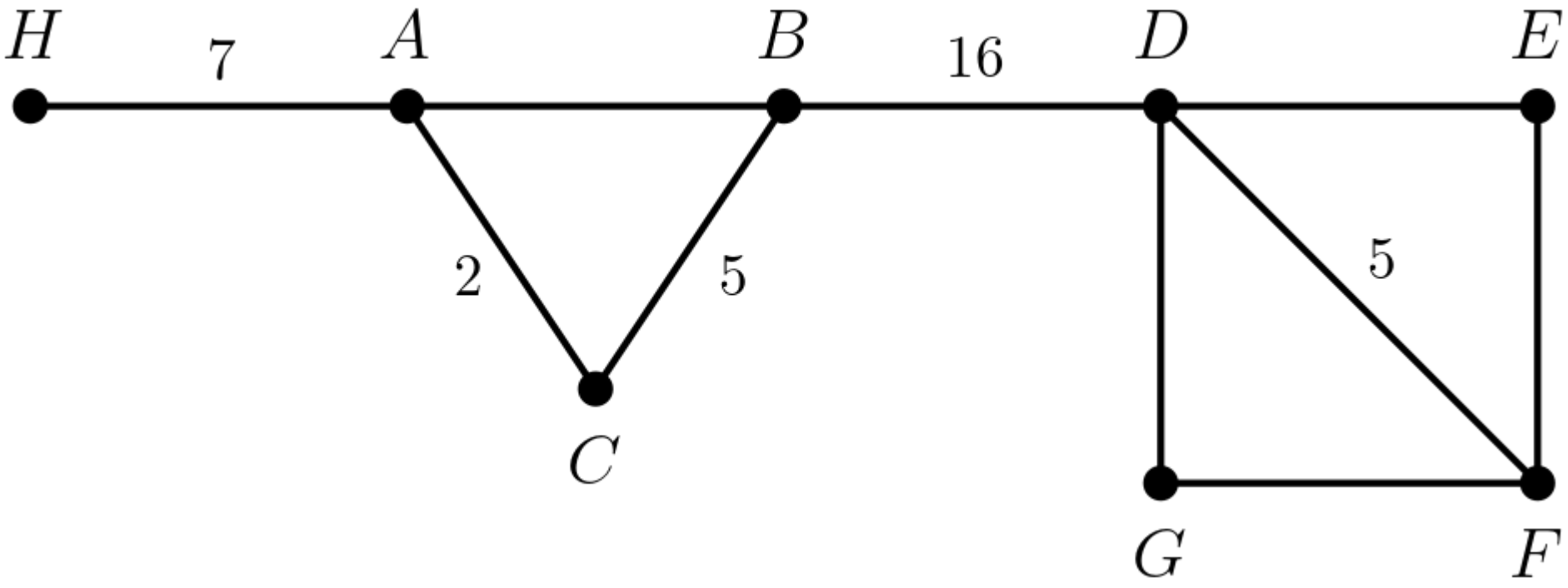
$$b(e) = \sum_{\substack{x,y \in V(G) \\ x \neq y}} \frac{\#\ shortest\ x - y\ paths\ that\ use\ e}{\#\ all\ shortest\ x - y\ paths}$$

This labelling is called the **betweenness centrality** (sometimes also just called betweenness) **of the edge e**. The betweenness (centrality) for vertices is defined analogously.

Idea: High betweenness indicates:

1. An edge/vertex of G where many paths must run through (hence, maybe low connectivity).
2. A central position for the edge/vertex, otherwise not many paths.

Betweenness centrality for edges



Divisive hierarchical clustering via betweenness: Girvan-Newman Algorithm

- Start with one cluster (if our initial graph G is **connected**).
- In each step, we consider a subgraph H of G and the clusters correspond to **connected components** of H.

Divisive hierarchical clustering via betweenness: Girvan-Newman Algorithm

- Start with one cluster (if our initial graph G is **connected**).
- In each step, we consider a subgraph H of G and the clusters correspond to **connected components** of H.
- Per step: **delete** the **edges** of **highest betweenness**.

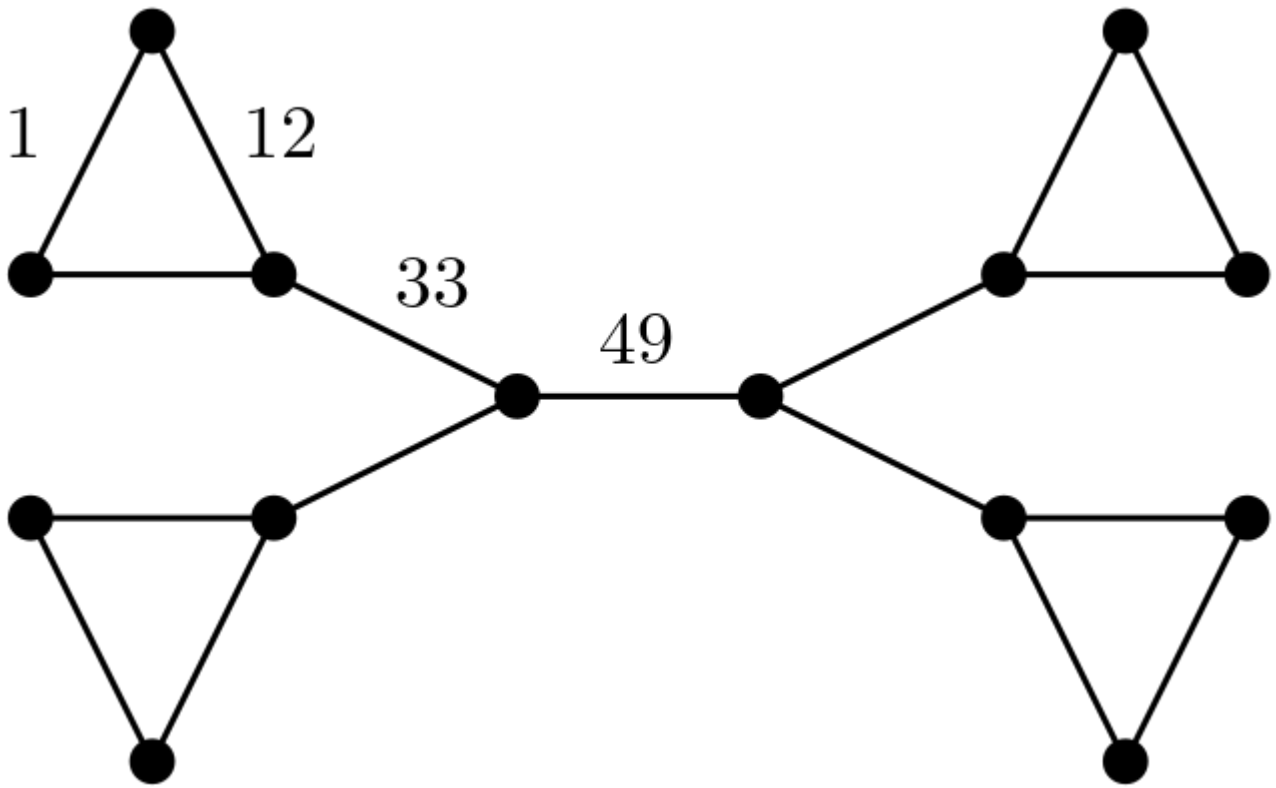
Divisive hierarchical clustering via betweenness: Girvan-Newman Algorithm

- Start with one cluster (if our initial graph G is **connected**).
- In each step, we consider a subgraph H of G and the clusters correspond to **connected components** of H.
- Per step: **delete** the **edges** of **highest betweenness**.
  - If some connected component / cluster decomposes into new ones (2 or maybe more), replace cluster by the new connected components (2 or maybe more).

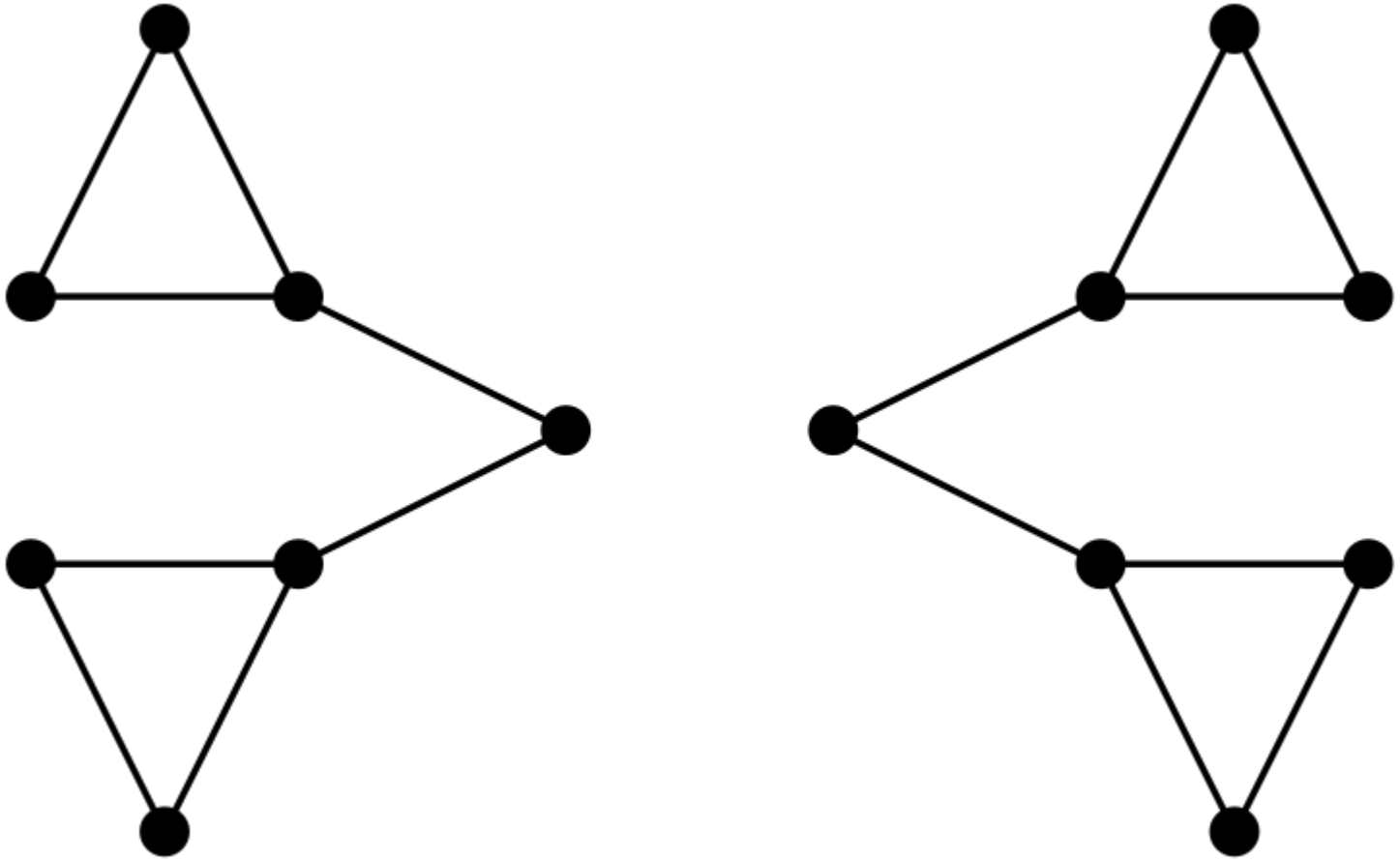
Divisive hierarchical clustering via betweenness: Girvan-Newman Algorithm

- Start with one cluster (if our initial graph G is **connected**).
- In each step, we consider a subgraph H of G and the clusters correspond to **connected components** of H.
- Per step: **delete** the **edges** of **highest betweenness**.
  - If some connected component / cluster decomposes into new ones (2 or maybe more), replace cluster by the new connected components (2 or maybe more).
  - If deletion of the edges does not disconnect a cluster, keep it.

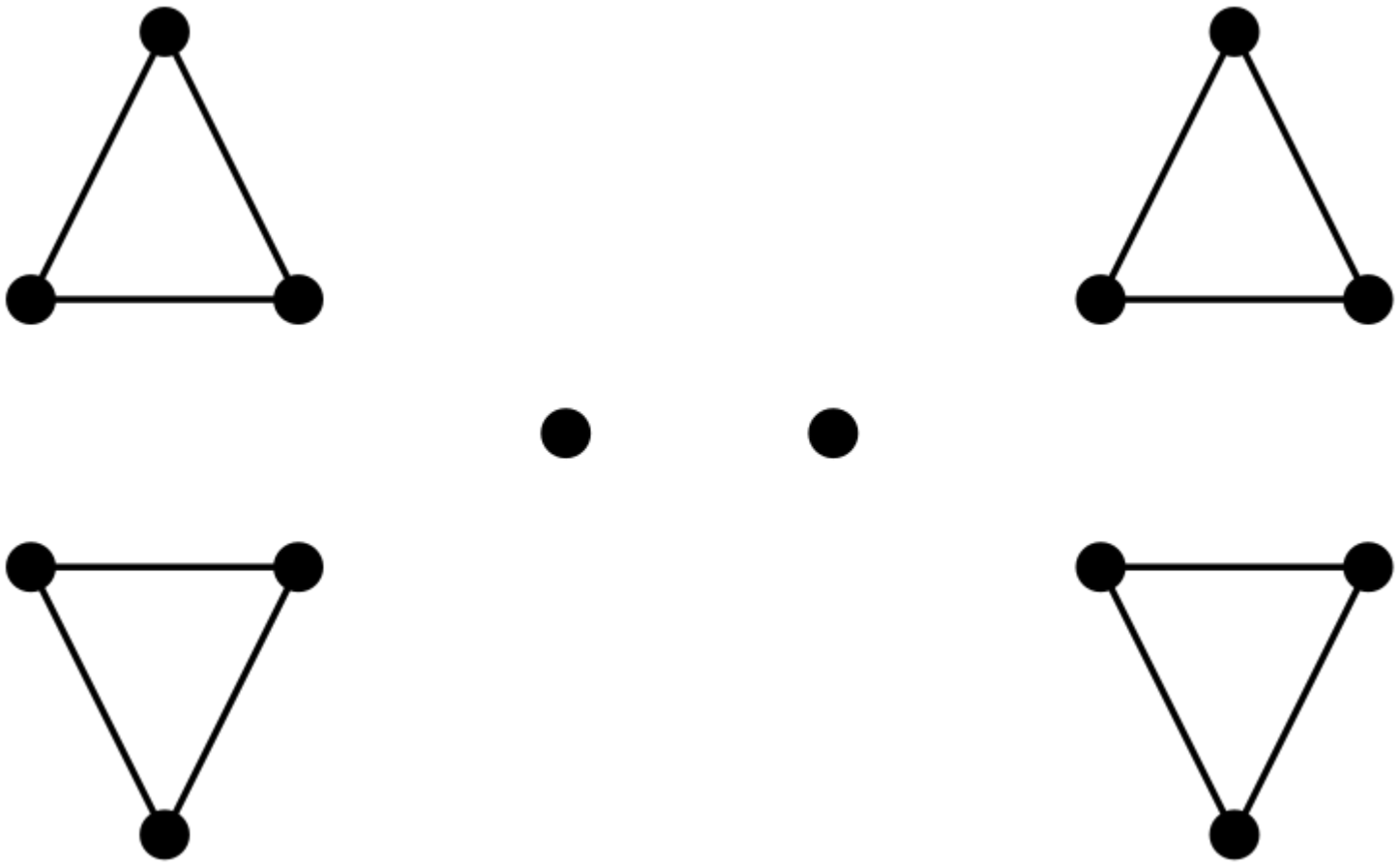
Divisive hierarchical clustering via betweenness: Girvan-Newman Algorithm



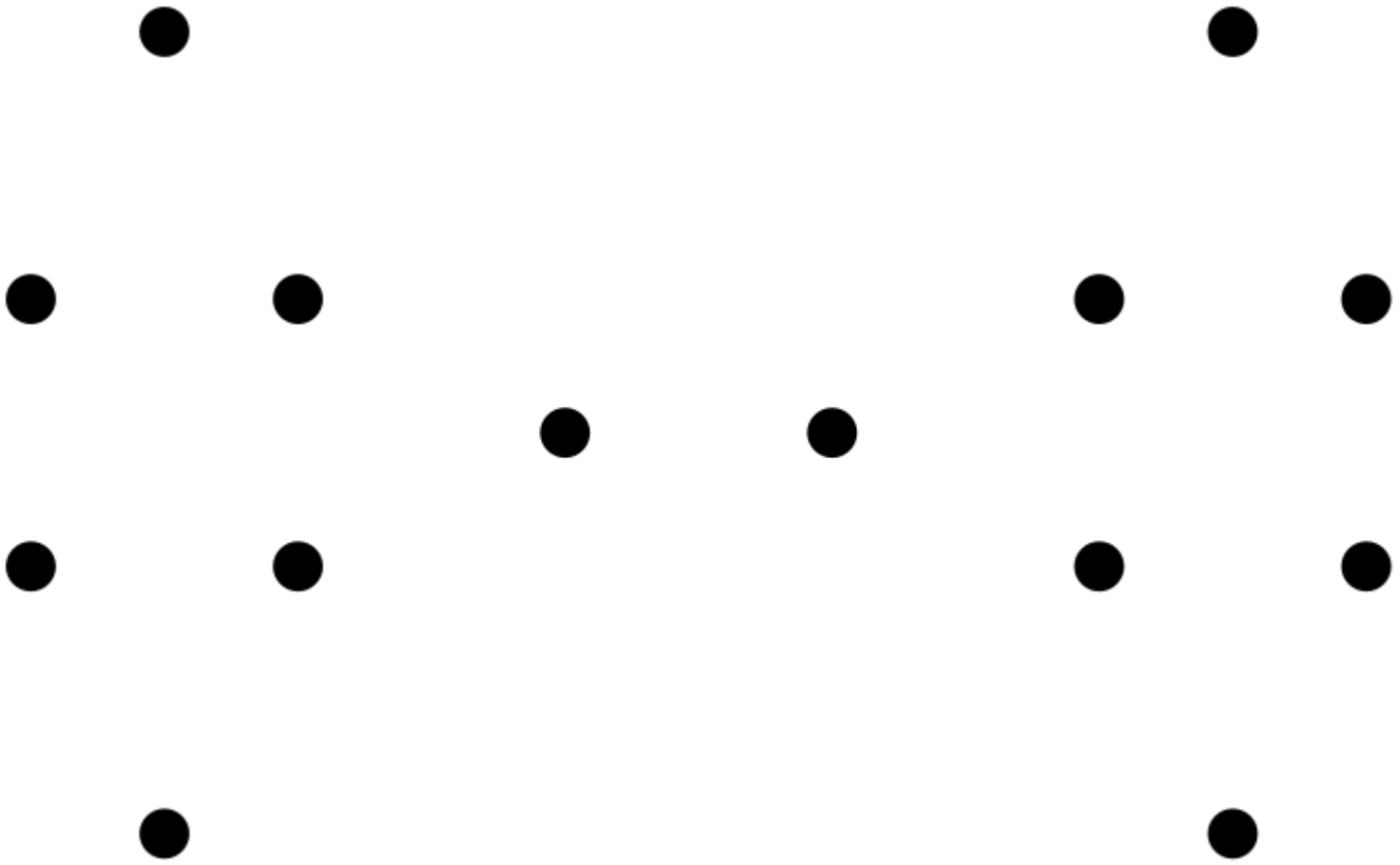
Divisive hierarchical clustering via betweenness: Girvan-Newman Algorithm



Divisive hierarchical clustering via betweenness: Girvan-Newman Algorithm



Divisive hierarchical clustering via betweenness: Girvan-Newman Algorithm



Computing betweenness for edges

- We need to count shortest paths (that use a specific edge).

Computing betweenness for edges

- We need to count shortest paths (that use a specific edge).

- **Girvan-Newman Algorithm**
  - Count shortest paths from a given start vertex.
    - This is done in 3 steps.

Computing betweenness for edges

- We need to count shortest paths (that use a specific edge).

- **Girvan-Newman Algorithm**
  - Count shortest paths from a given start vertex.
    - This is done in 3 steps.
  - We repeat this for every vertex.
    - Eventually, we will have counted each shortest path twice.

Computing betweenness for edges: Step 1

- Let G be the given graph with $|V(G)| = n$ and $|E(G)| = m$ for some $m, n \in \mathbb{N}$. We **fix a vertex** of G, call it $\boldsymbol{r}$.

Computing betweenness for edges: Step 1

- Let G be the given graph with $|V(G)| = n$ and $|E(G)| = m$ for some $m, n \in \mathbb{N}$. We **fix a vertex** of G, call it $\boldsymbol{r}$.
- Perform a **breadth-first search** (BFS) in G with $\boldsymbol{r}$ **as** the **root**.

Computing betweenness for edges: Step 1

- Let G be the given graph with $|V(G)| = n$ and $|E(G)| = m$ for some $m, n \in \mathbb{N}$. We **fix a vertex** of G, call it $\boldsymbol{r}$.
- Perform a **breadth-first search** (BFS) in G with $\boldsymbol{r}$ **as** the **root**.
- **Instead** of storing a **breadth-first search tree**, we store precisely all those edges that lie on some shortest path starting in r.

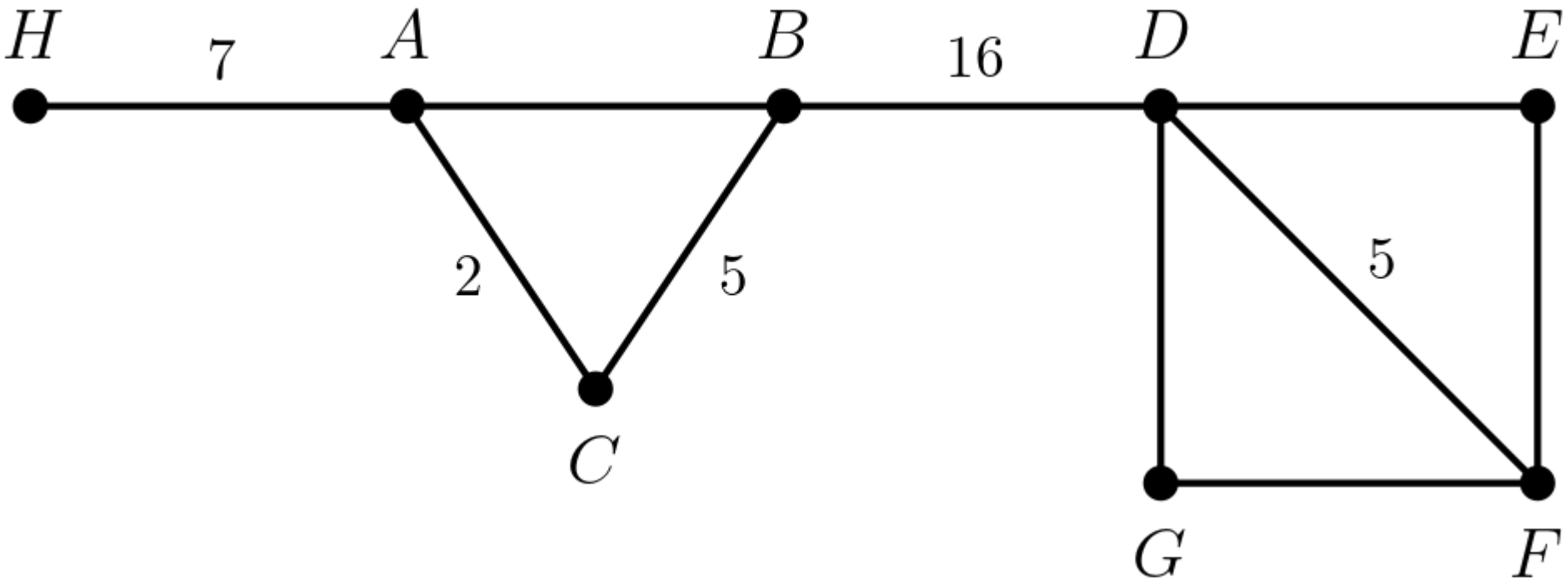
Computing betweenness for edges: Step 1

- Let G be the given graph with $|V(G)| = n$ and $|E(G)| = m$ for some $m, n \in \mathbb{N}$. We **fix a vertex** of G, call it $\boldsymbol{r}$.
- Perform a **breadth-first search** (BFS) in G with $\boldsymbol{r}$ **as** the **root**.
- **Instead** of storing a **breadth-first search tree**, we store precisely all those edges that lie on some shortest path starting in r.
- In other words: we store **no edges** whose end vertices lie in the **same distance class** w.r.t. r, but **all edges** whose end vertices lie in **different distance classes** w.r.t. r.

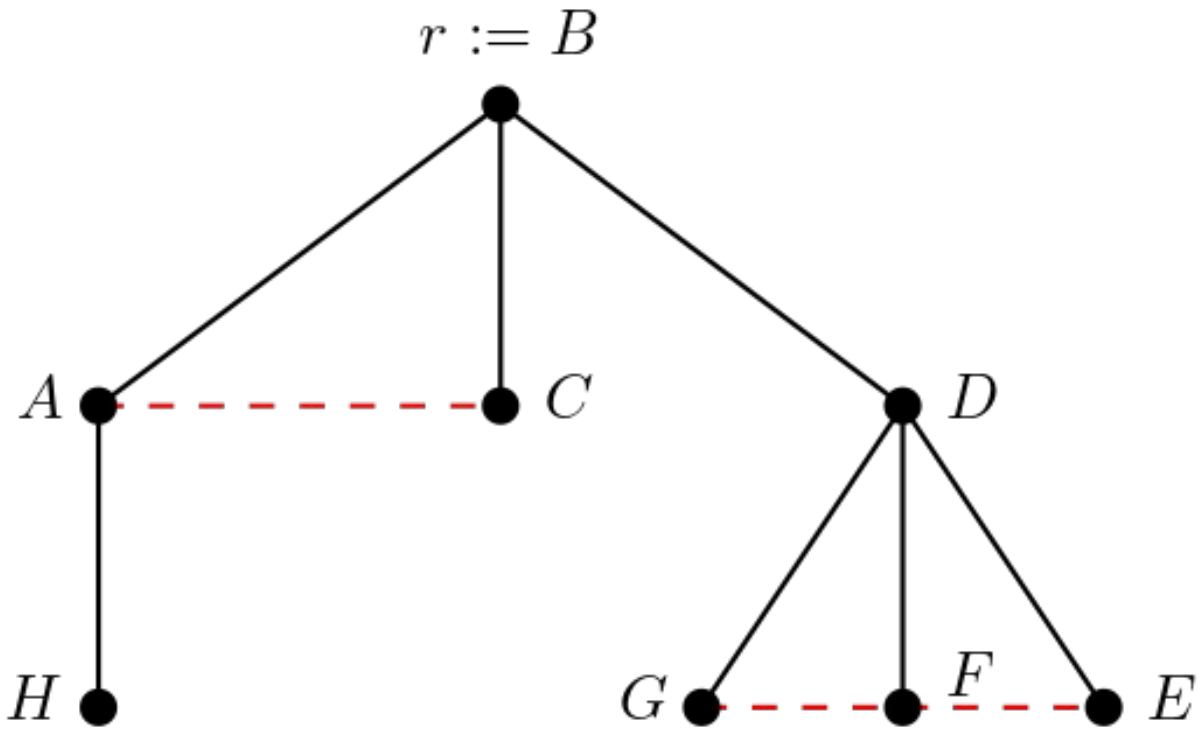
Computing betweenness for edges: Step 1

- Let G be the given graph with $|V(G)| = n$ and $|E(G)| = m$ for some $m, n \in \mathbb{N}$. We **fix a vertex** of G, call it $\boldsymbol{r}$.
- Perform a **breadth-first search** (BFS) in G with $\boldsymbol{r}$ **as** the **root**.
- **Instead** of storing a **breadth-first search tree**, we store precisely all those edges that lie on some shortest path starting in r.
- In other words: we store **no edges** whose end vertices lie in the **same distance class** w.r.t. r, but **all edges** whose end vertices lie in **different distance classes** w.r.t. r.
- The **running time** of BFS is $\boldsymbol{O(m)}$.

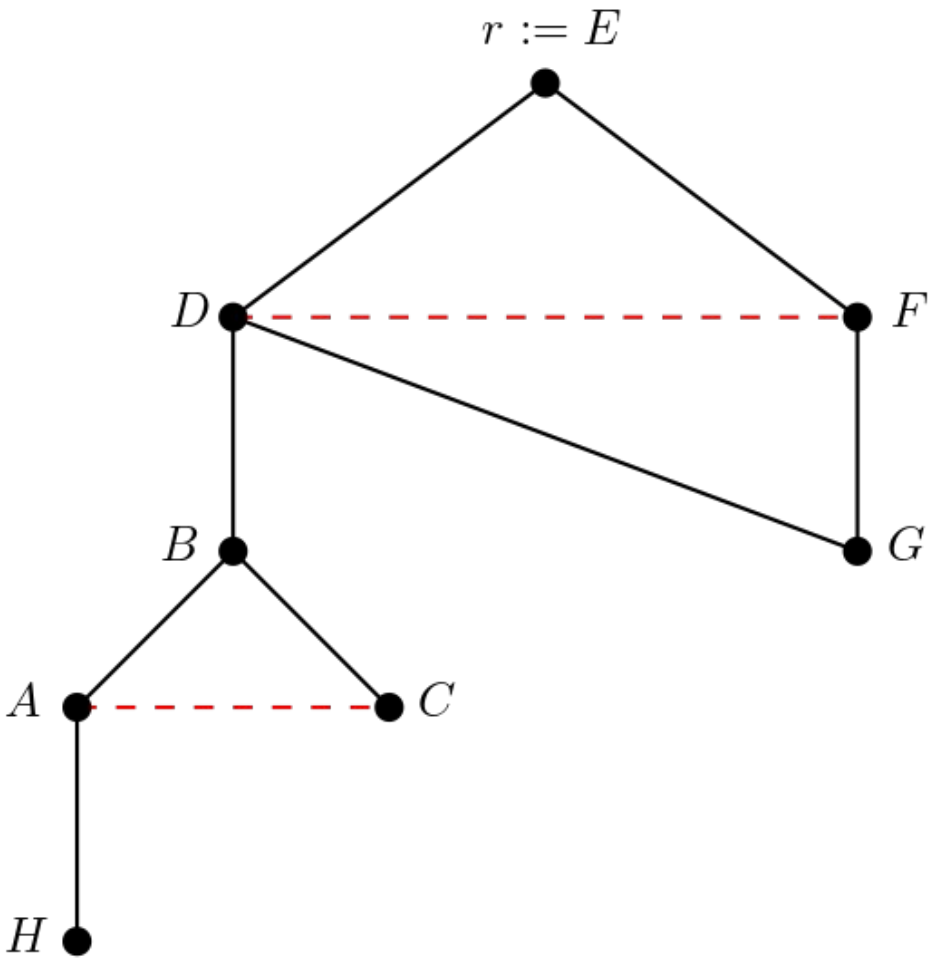
Computing betweenness for edges: Step 1



Computing betweenness for edges: Step 1



Computing betweenness for edges: Step 1



Computing betweenness for edges: Step 2

For each vertex v, count # of shortest paths from r ending in v.

Computing betweenness for edges: Step 2

For each vertex v, count # of shortest paths from r ending in v.

- Label r with $\ell_2(r) := 1$. (Note that r has distance 0 to itself.)

Computing betweenness for edges: Step 2

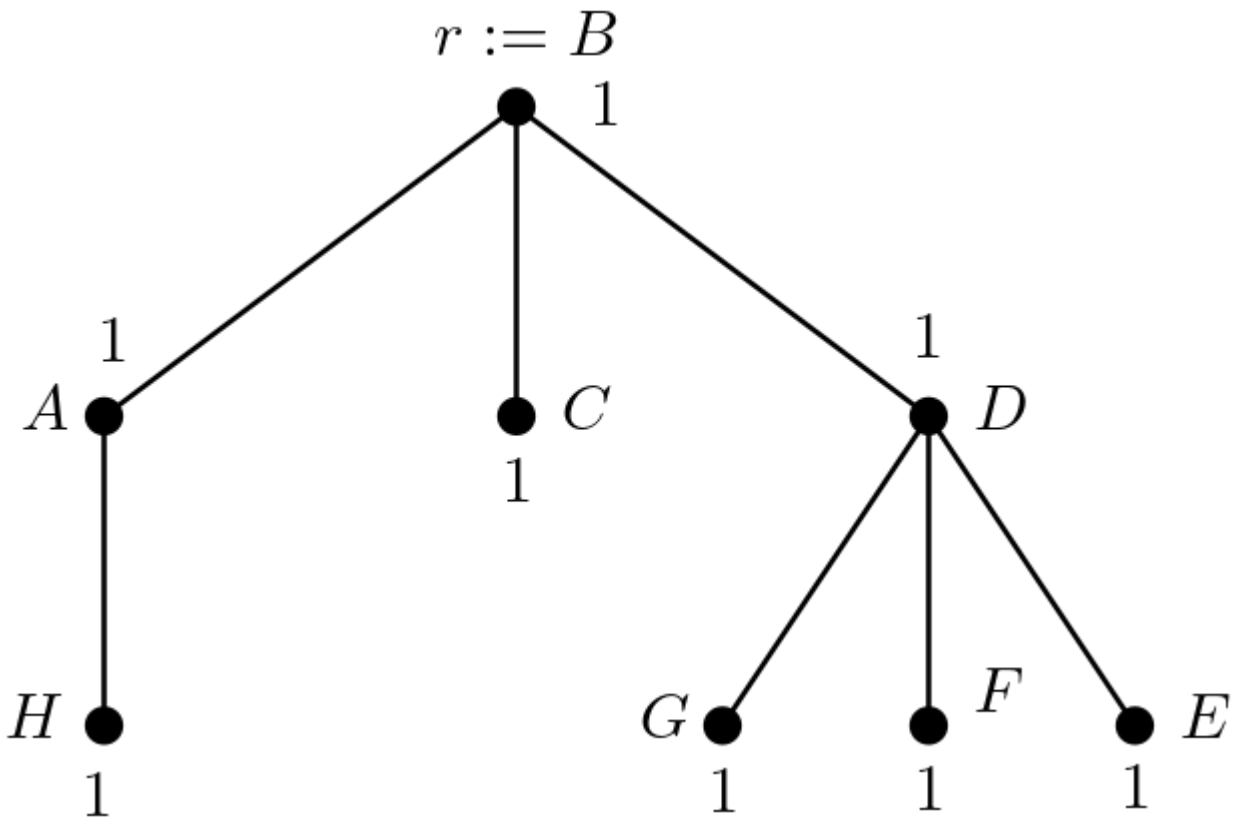
For each vertex v, count # of shortest paths from r ending in v.

- Label r with $\ell_2(r) := 1$. (Note that r has distance 0 to itself.)
- Let v be a vertex in distance class D_i for $i > 0$. Now label v with:

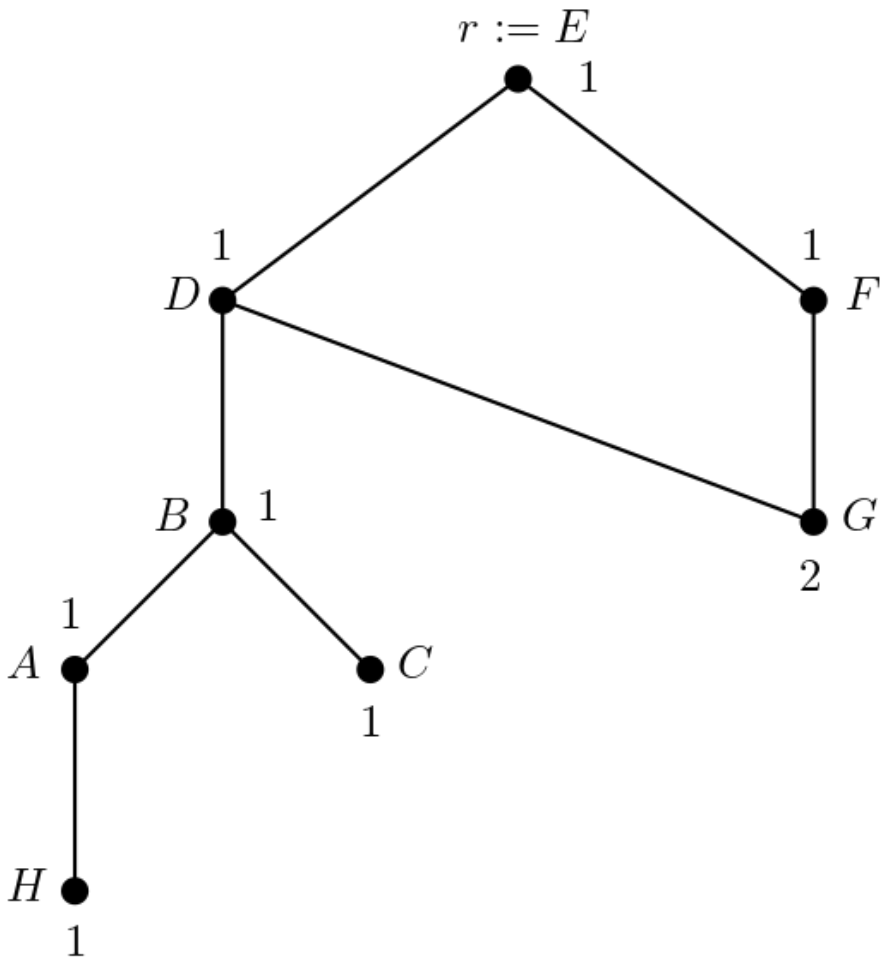
$$\ell_2(v) := \sum_{\substack{w \in D_{i-1} \\ vw \in E(G)}} \ell_2(w)$$

Hence, we label top (from root r) down along the distance classes.

Computing betweenness for edges: Step 2



Computing betweenness for edges: Step 2



Computing betweenness for edges: Step 3

Label each edge e with the following count $b_r(e)$:

$$b_r(e) = \sum_{y \in V(G)} \frac{\#\ shortest\ r - y\ paths\ that\ use\ e}{\#\ all\ shortest\ r - y\ paths}$$

To label all edges with this count, we use another auxiliary labelling ℓ_3 for the vertices.

Computing betweenness for edges: Step 3

Intuition:

- Think of ℓ_3 and b_r as demand and flow.
- **BUT** instead of Kirchhoff's law, each vertex consumes a flow of 1.

Computing betweenness for edges: Step 3

Intuition:

- Think of ℓ_3 and b_r as flows.
- **BUT** instead of Kirchhoff's law, each vertex consumes a flow of 1.
- We demand that each vertex without neighbours in later distance classes receives a flow of 1.

Computing betweenness for edges: Step 3

Intuition:

- Think of ℓ_3 and b_r as flows.
- **BUT** instead of Kirchhoff's law, each vertex consumes a flow of 1.
- We demand that each vertex without neighbours in later distance classes receives a flow of 1.
- Each other vertex must receive (1 + what is sent further).

Computing betweenness for edges: Step 3

Intuition:

- Think of ℓ_3 and b_r as demand and flow.
- **BUT** instead of Kirchhoff's law, each vertex consumes a flow of 1.
- We demand that each vertex without neighbours in later distance classes receives a flow of 1.
- Each other vertex must receive (1 + what is sent further).
- How is the total flow (-1) which enters a vertex w split among the edges vw that enter w:
  - via the fraction $\frac{\ell_2(v)}{\ell_2(w)}$.

Girvan-Newman Algorithm: Step 3

- Label each vertex v that has no neighbour in a later distance class with: $\ell_3(v) := 1$.

Girvan-Newman Algorithm: Step 3

- Label each vertex v that has no neighbour in a later distance class with: $\ell_3(v) \coloneqq 1$.
- Let v be a vertex in distance class D_i for $i > 0$ with a neighbour in D_{i+1}. Now label v with:

$$\ell_3(v) \coloneqq 1 + \sum_{\substack{w \in D_{i+1} \\ vw \in E(G)}} b_r(vw)$$

Girvan-Newman Algorithm: Step 3

- Label each vertex v that has no neighbour in a later distance class with: $\ell_3(v) := 1$.
- Let v be a vertex in distance class D_i for $i > 0$ with a neighbour in D_{i+1}. Now label v with:

$$\ell_3(v) := 1 + \sum_{\substack{w \in D_{i+1} \\ vw \in E(G)}} b_r(vw)$$

- Let $w \in D_{i+1}$ and let $v_1, \dots, v_k$ be the neighbours of w in D_i. Let ℓ_2 denote the labelling we have assigned in Step 2. Now label each edge $v_j w$ by

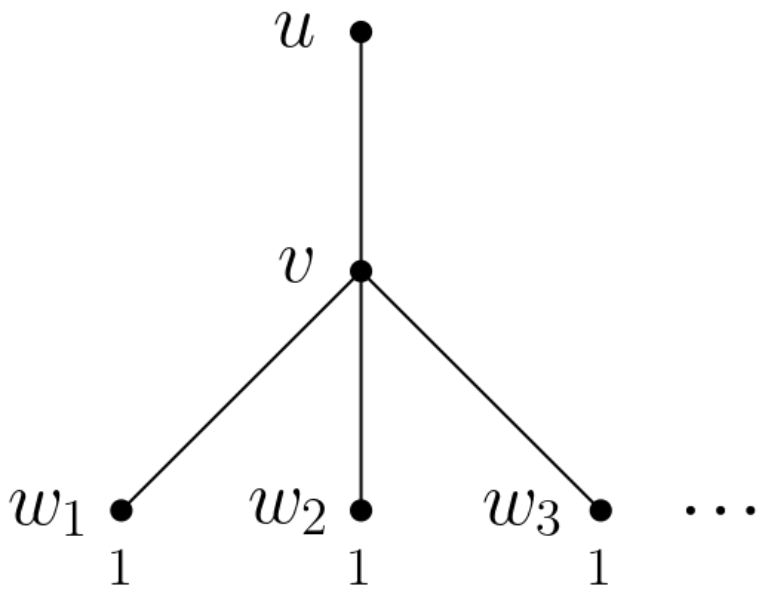
$$b_r(v_j w) := \ell_3(w) \cdot \frac{\ell_2(v_j)}{\sum_{p=1}^{k} \ell_2(v_p)} = \ell_3(w) \cdot \frac{\ell_2(v_j)}{\ell_2(w)}$$

Computing betweenness for edges: Step 3

- Let $\ell_3(w_i) = 1$.

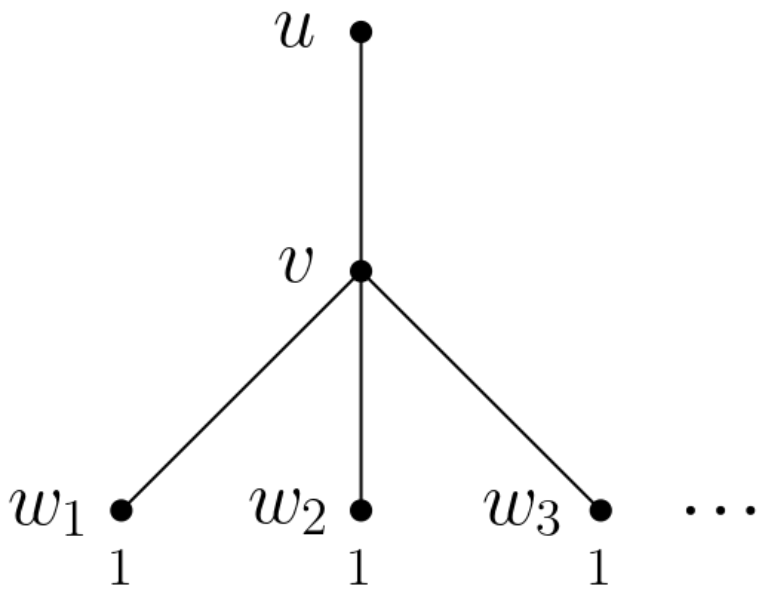
- $b_r(vw_i) = 1 \cdot \frac{\ell_2(v)}{\ell_2(w_i)}$

So $b_r(vw_i)$ really counts the fraction of all r-w_i paths that use the edge vw_i.



Computing betweenness for edges: Step 3

- Let $\ell_3(w_i) = 1$.
- $b_r(uv) = \ell_3(v) \cdot \frac{\ell_2(u)}{\ell_2(v)} =$

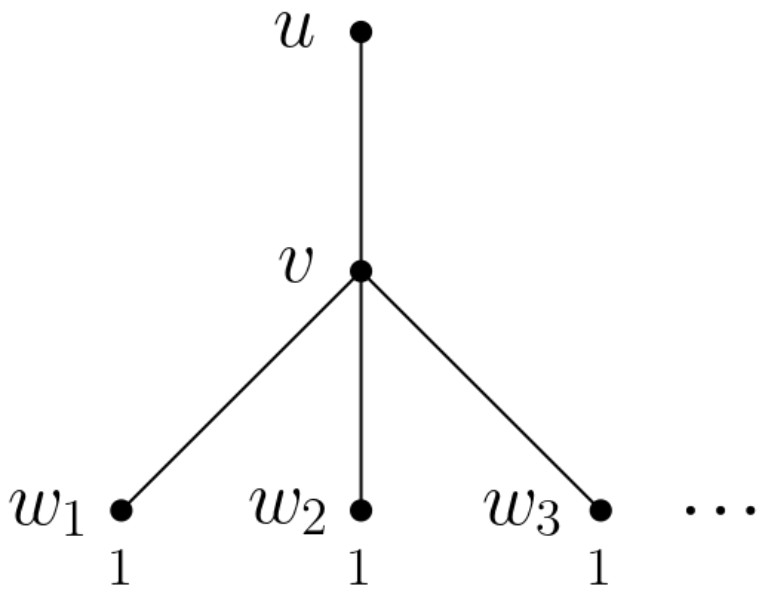


Computing betweenness for edges: Step 3

- Let $\ell_3(w_i) = 1$.

- $b_r(uv) = \ell_3(v) \cdot \frac{\ell_2(u)}{\ell_2(v)} =$

$$= \frac{\ell_2(u)}{\ell_2(v)} + \frac{\ell_2(u)}{\ell_2(v)} \sum_{w_i} b_r(vw_i) =$$



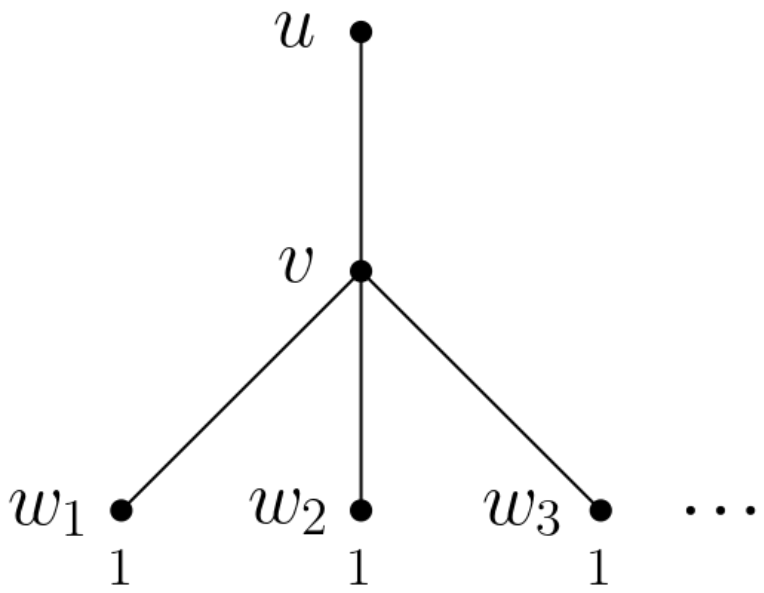
Computing betweenness for edges: Step 3

- Let $\ell_3(w_i) = 1$.

- $b_r(uv) = \ell_3(v) \cdot \frac{\ell_2(u)}{\ell_2(v)} =$

$$= \frac{\ell_2(u)}{\ell_2(v)} + \frac{\ell_2(u)}{\ell_2(v)} \sum_{w_i} b_r(vw_i) =$$

$$= \frac{\ell_2(u)}{\ell_2(v)} + \frac{\ell_2(u)}{\ell_2(v)} \sum_{w_i} \frac{\ell_2(v)}{\ell_2(w_i)} =$$



Computing betweenness for edges: Step 3

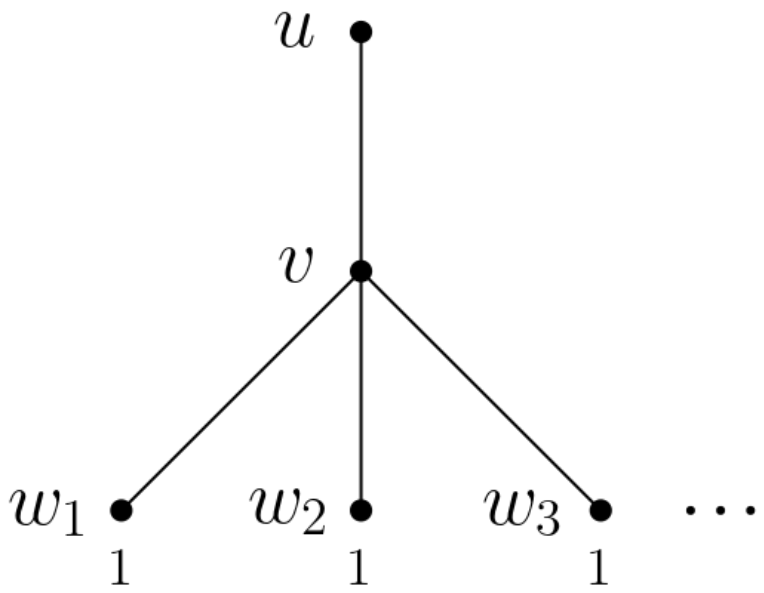
- Let $\ell_3(w_i) = 1$.

- $b_r(uv) = \ell_3(v) \cdot \frac{\ell_2(u)}{\ell_2(v)} =$

$$= \frac{\ell_2(u)}{\ell_2(v)} + \frac{\ell_2(u)}{\ell_2(v)} \sum_{w_i} b_r(vw_i) =$$

$$= \frac{\ell_2(u)}{\ell_2(v)} + \frac{\ell_2(u)}{\ell_2(v)} \sum_{w_i} \frac{\ell_2(v)}{\ell_2(w_i)} =$$

$$= \frac{\ell_2(u)}{\ell_2(v)} + \sum_{w_i} \frac{\ell_2(u)}{\ell_2(w_i)}$$



Computing betweenness for edges: Step 3

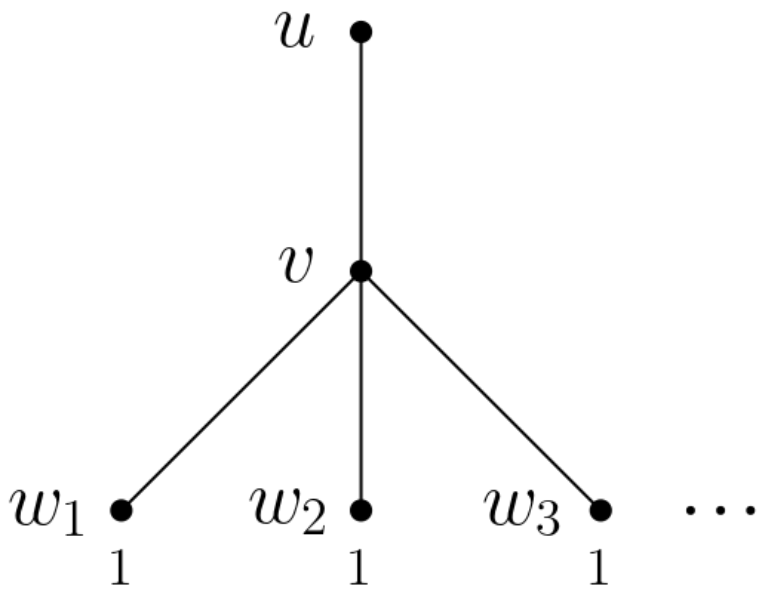
- Let $\ell_3(w_i) = 1$.

- $b_r(uv) = \ell_3(v) \cdot \frac{\ell_2(u)}{\ell_2(v)} =$

$$= \frac{\ell_2(u)}{\ell_2(v)} + \frac{\ell_2(u)}{\ell_2(v)} \sum_{w_i} b_r(vw_i) =$$

$$= \frac{\ell_2(u)}{\ell_2(v)} + \frac{\ell_2(u)}{\ell_2(v)} \sum_{w_i} \frac{\ell_2(v)}{\ell_2(w_i)} =$$

$$= \frac{\ell_2(u)}{\ell_2(v)} + \sum_{w_i} \frac{\ell_2(u)}{\ell_2(w_i)}$$



So $b_r(uv)$ really counts the fraction of all shortest paths from r that use uv.

Computing betweenness for edges

- In order to compute the betweenness centrality for an edge e, we have to perform the three previous steps for every vertex as root.

Computing betweenness for edges

- In order to compute the betweenness centrality for an edge e, we have to perform the three previous steps for every vertex as root.
- Set $b_r(e) = 0$ if e is not contained in the BFS structure.

Computing betweenness for edges

- In order to compute the betweenness centrality for an edge e, we have to perform the three previous steps for every vertex as root.
- Set $b_r(e) = 0$ if e is not contained in the BFS structure.
- Then we get:

$$b(e) = \frac{1}{2} \sum_{r \in V(G)} b_r(e)$$

Computing betweenness for edges

- In order to compute the betweenness centrality for an edge e, we have to perform the three previous steps for every vertex as root.
- Set $b_r(e) = 0$ if e is not contained in the BFS structure.
- Then we get:

$$b(e) = \frac{1}{2} \sum_{r \in V(G)} b_r(e)$$

- Note that we need to divide by 2 since we count each shortest path twice, namely for each of the two ways of how to traverse the path.

Computing betweenness for edges

- In order to compute the betweenness centrality for an edge e, we have to perform the three previous steps for every vertex as root.
- Set $b_r(e) = 0$ if e is not contained in the BFS structure.
- Then we get:

$$b(e) = \frac{1}{2} \sum_{r \in V(G)} b_r(e)$$

- Note that we need to divide by 2 since we count each shortest path twice, namely for each of the two ways of how to traverse the path.
- The running time for the betweenness computation is $O(nm)$. So if our graph is not sparse, then $O(nm) = O(n^3)$.

Computing betweenness for edges

- In order to compute the betweenness centrality for an edge e, we have to perform the three previous steps for every vertex as root.
- Set $b_r(e) = 0$ if e is not contained in the BFS structure.
- Then we get:

$$b(e) = \frac{1}{2} \sum_{r \in V(G)} b_r(e)$$

- Note that we need to divide by 2 since we count each shortest path twice, namely for each of the two ways of how to traverse the path.
- The running time for the betweenness computation is $O(nm)$. So if our graph is not sparse, then $O(nm) = O(n^3)$.
- For huge data sets: only take random set of vertices for BFS roots.

Modularity

Quality of clusters / communities: Modularity

- Given a graph G and a partitioning (clustering) $\mathcal{C} = (C_1, \dots, C_k)$ of $V(G)$ into communities (clusters) C_i.
- **Rough idea:** Define ***modularity*** $Q(G, \mathcal{C})$ as a measure via:

$$\sum_{C_i \in \mathcal{C}} [\, (\#\ of\ edges\ within\ C_i) - (\boldsymbol{expected}\ \#\ of\ edges\ within\ C_i)\,]$$

- We need a model for an average / random graph on our cluster C_i.

Model for a random graph

- Let G be a graph. Define a **random (multi)graph** $\mathcal{R}_d(G)$ on the same vertex set which keeps all **vertex degrees** (# edges incident with a vertex)

Model for a random graph

- Let G be a graph. Define a **random (multi)graph** $\mathcal{R}_d(G)$ on the same vertex set which keeps all **vertex degrees** (# edges incident with a vertex), i.e.:

1. $V\big(\mathcal{R}_d(G)\big) = V(G)$.

Model for a random graph

- Let G be a graph. Define a **random (multi)graph** $\mathcal{R}_d(G)$ on the same vertex set which keeps all **vertex degrees** (# edges incident with a vertex), i.e.:

1. $V\big(\mathcal{R}_d(G)\big) = V(G)$.
2. $d_{\mathcal{R}_d(G)}(v) = d_G(v)$ for every $v \in V(G)$.

Model for a random graph

- Let G be a graph. Define a **random (multi)graph** $\mathcal{R}_d(G)$ on the same vertex set which keeps all **vertex degrees** (# edges incident with a vertex), i.e.:

1. $V\big(\mathcal{R}_d(G)\big) = V(G)$.
2. $d_{\mathcal{R}_d(G)}(v) = d_G(v)$ for every $v \in V(G)$.

- Here let $d_G(v)$ denote the # edges incident with vertex v in G. Hence, $d_{\mathcal{R}_d(G)}(v)$ denotes the # edges incident with v in $\mathcal{R}_d(G)$.

Model for a random graph

- Let G be a graph. Define a **random (multi)graph** $\mathcal{R}_d(G)$ on the same vertex set which keeps all **vertex degrees** (# edges incident with a vertex), i.e.:

1. $V\big(\mathcal{R}_d(G)\big) = V(G)$.
2. $d_{\mathcal{R}_d(G)}(v) = d_G(v)$ for every $v \in V(G)$.

- Here let $d_G(v)$ denote the # edges incident with vertex v in G. Hence, $d_{\mathcal{R}_d(G)}(v)$ denotes the # edges incident with v in $\mathcal{R}_d(G)$.

- Idea of building $\mathcal{R}_d(G)$: Cut the edges at every vertex and rewire them randomly, allowing multiple edges and loops.

Model for a random graph

- Let $v, w \in V(G)$ and let $|E(G)| = m$. We enumerate the edges incident with v within G from 1 to $d_G(v)$.

Model for a random graph

- Let $v, w \in V(G)$ and let $|E(G)| = m$. We enumerate the edges incident with v within G from 1 to $d_G(v)$.

- Probability that the i-th edge incident with v is rewired to some edge incident with w:

$$\frac{d_G(w)}{2m - 1}$$

Model for a random graph

- Let $v, w \in V(G)$ and let $|E(G)| = m$. We enumerate the edges incident with v within G from 1 to $d_G(v)$.

- Probability that the i-th edge incident with v is rewired to some edge incident with w:

$$\frac{d_G(w)}{2m-1}$$

- Expected # of edges between v and w in $\mathcal{R}_d(G)$:

$$d_G(v) \cdot \frac{d_G(w)}{2m-1} \approx \frac{d_G(v) \cdot d_G(w)}{2m}$$

Quality of clusters / communities: Modularity

- Given a graph G with $|E(G)| = m$ and a clustering $\mathcal{C} = (C_1, \dots, C_k)$ of $V(G)$ into communities C_i.
- Define **modularity** $Q(G, C)$ as:

$$Q(G, \mathcal{C}) = \frac{1}{2m} \cdot \sum_{C_i \in \mathcal{C}} \sum_{\substack{v,w \in V(C_i) \\ v \neq w}} \left[A_{vw} - \frac{d_G(v) \cdot d_G(w)}{2m} \right]$$

Quality of clusters / communities: Modularity

- Given a graph G with $|E(G)| = m$ and a clustering $\mathcal{C} = (C_1, \dots, C_k)$ of $V(G)$ into communities C_i.
- Define **modularity** $Q(G, C)$ as:

$$Q(G, \mathcal{C}) = \frac{1}{2m} \cdot \sum_{C_i \in \mathcal{C}} \sum_{\substack{v,w \in V(C_i) \\ v \neq w}} \left[A_{vw} - \frac{d_G(v) \cdot d_G(w)}{2m} \right]$$

- Here A_{vw} is the indicator function: $A_{vw} = \begin{cases} 1, & if\ vw \in E(C_i) \\ 0, & if\ vw \notin E(C_i) \end{cases}$

Quality of clusters / communities: Modularity

- Given a graph G with $|E(G)| = m$ and a clustering $\mathcal{C} = (C_1, \dots, C_k)$ of $V(G)$ into communities C_i.
- Define **modularity** $Q(G, C)$ as:

$$Q(G, \mathcal{C}) = \frac{1}{2m} \cdot \sum_{C_i \in \mathcal{C}} \sum_{\substack{v,w \in V(C_i) \\ v \neq w}} \left[A_{vw} - \frac{d_G(v) \cdot d_G(w)}{2m} \right]$$

- Here A_{vw} is the indicator function: $A_{vw} = \begin{cases} 1, & if\ vw \in E(C_i) \\ 0, & if\ vw \notin E(C_i) \end{cases}$
- We scale the sum by $\frac{1}{2m}$ to get the range for $Q(G, \mathcal{C})$ within $[-1, 1]$.

Quality of clusters / communities: Modularity

- Given a graph G with $|E(G)| = m$ and a clustering $\mathcal{C} = (C_1, \dots, C_k)$ of $V(G)$ into communities C_i.
- Define **modularity** $Q(G, C)$ as:

$$Q(G, \mathcal{C}) = \frac{1}{2m} \cdot \sum_{C_i \in \mathcal{C}} \sum_{\substack{v,w \in V(C_i) \\ v \neq w}} \left[A_{vw} - \frac{d_G(v) \cdot d_G(w)}{2m} \right]$$

- Here A_{vw} is the indicator function: $A_{vw} = \begin{cases} 1, & if\ vw \in E(C_i) \\ 0, & if\ vw \notin E(C_i) \end{cases}$
- We scale the sum by $\frac{1}{2m}$ to get the range for $Q(G, \mathcal{C})$ within $[-1, 1]$.
- Usually: $Q(G, \mathcal{C}) > 0{,}3$ is an indicator for good community structure.

Louvain algorithm

Louvain algorithm

- The Louvain algorithm is an agglomerative clustering algorithm.

Louvain algorithm

- The Louvain algorithm is an agglomerative clustering algorithm.
- The algorithm merges clusters by deciding locally where modularity is optimised the most.

Louvain algorithm

- The Louvain algorithm is an agglomerative clustering algorithm.
- The algorithm merges clusters by deciding locally where modularity is optimised the most.
- The algorithm is heuristic and might end in a local maximum.

Louvain algorithm

- The Louvain algorithm is an agglomerative clustering algorithm.
- The algorithm merges clusters by deciding locally where modularity is optimised the most.
- The algorithm is heuristic and might end in a local maximum.
- The running time of the algorithm is $O(m)$.

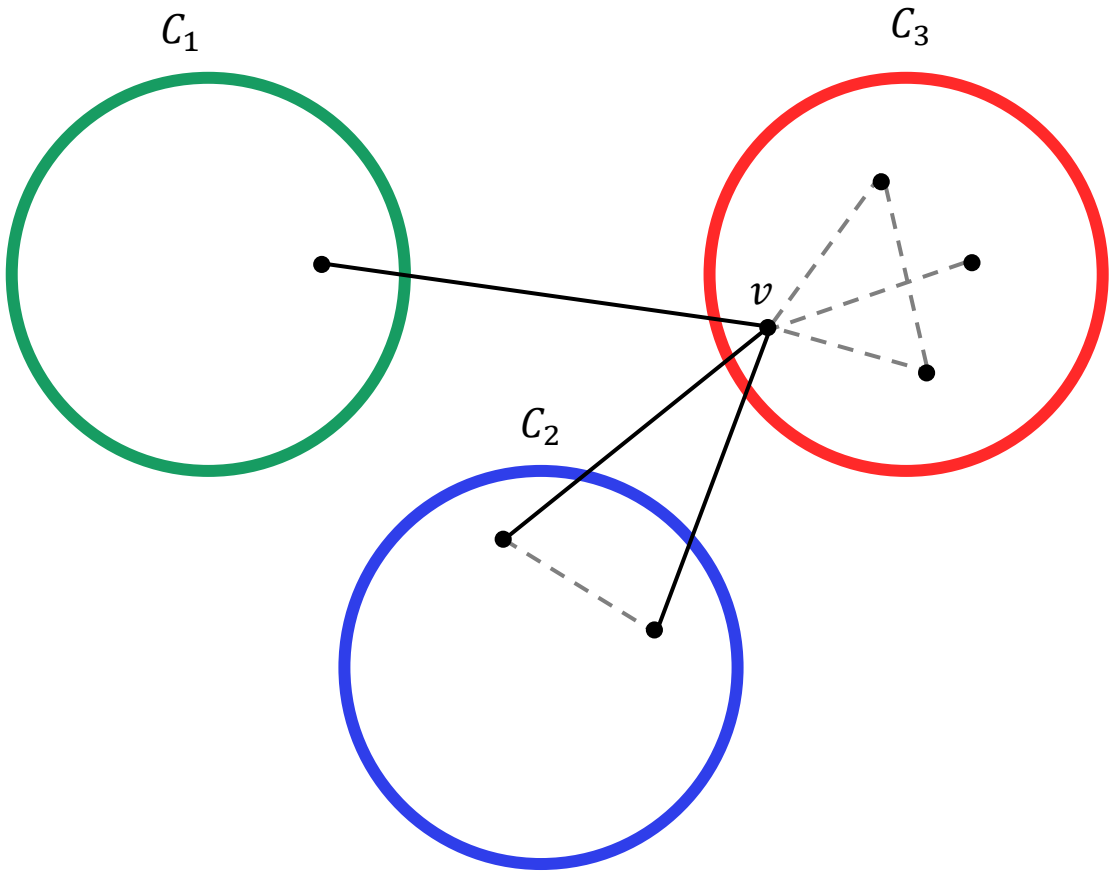
Louvain algorithm

- The Louvain algorithm is an agglomerative clustering algorithm.
- The algorithm merges clusters by deciding locally where modularity is optimised the most.
- The algorithm is heuristic and might end in a local maximum.
- The running time of the algorithm is $O(m)$.
  - Running time via considering random neighbours: $O(n \cdot \log(n))$

Louvain algorithm

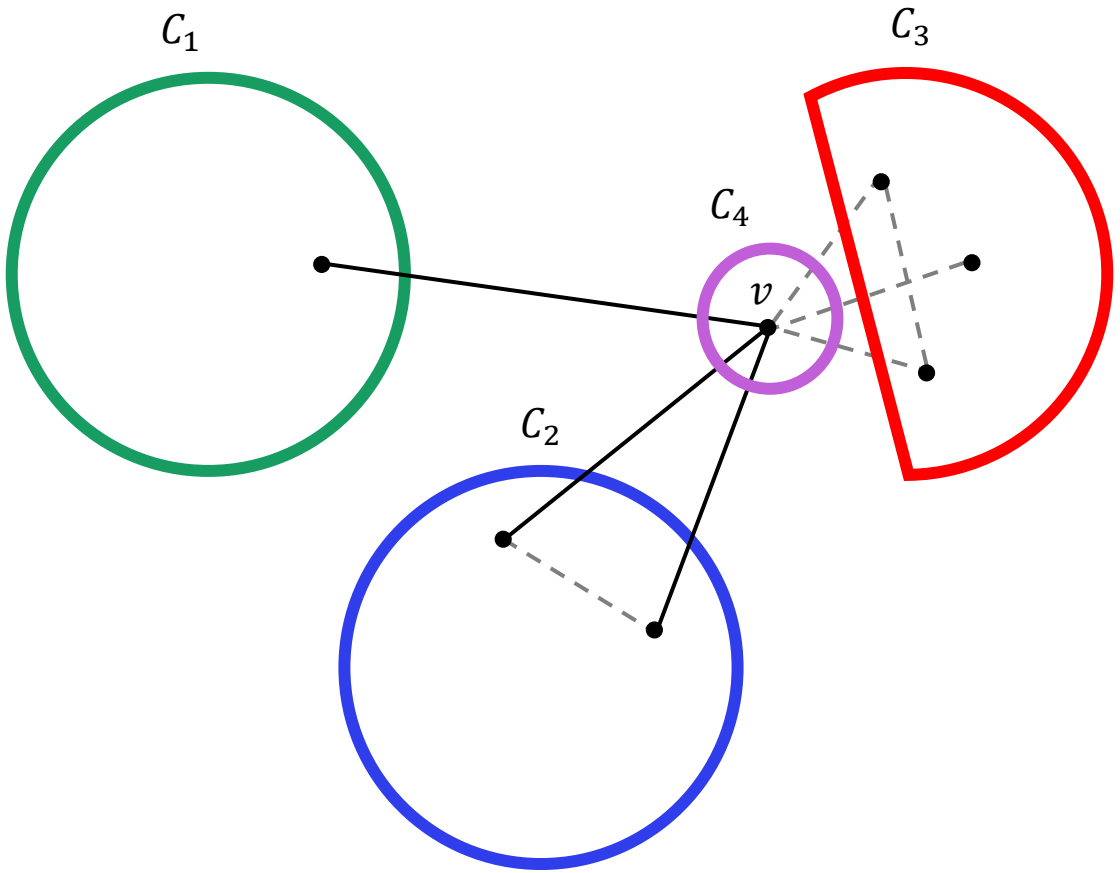
- The Louvain algorithm is an agglomerative clustering algorithm.
- The algorithm merges clusters by deciding locally where modularity is optimised the most.
- The algorithm is heuristic and might end in a local maximum.
- The running time of the algorithm is $O(m)$.
  - Running time via considering random neighbours: $O(n \log(n))$
- The algorithm can be split into 2 steps, which are iterated.

Louvain algorithm: the main idea



Louvain algorithm: the main idea

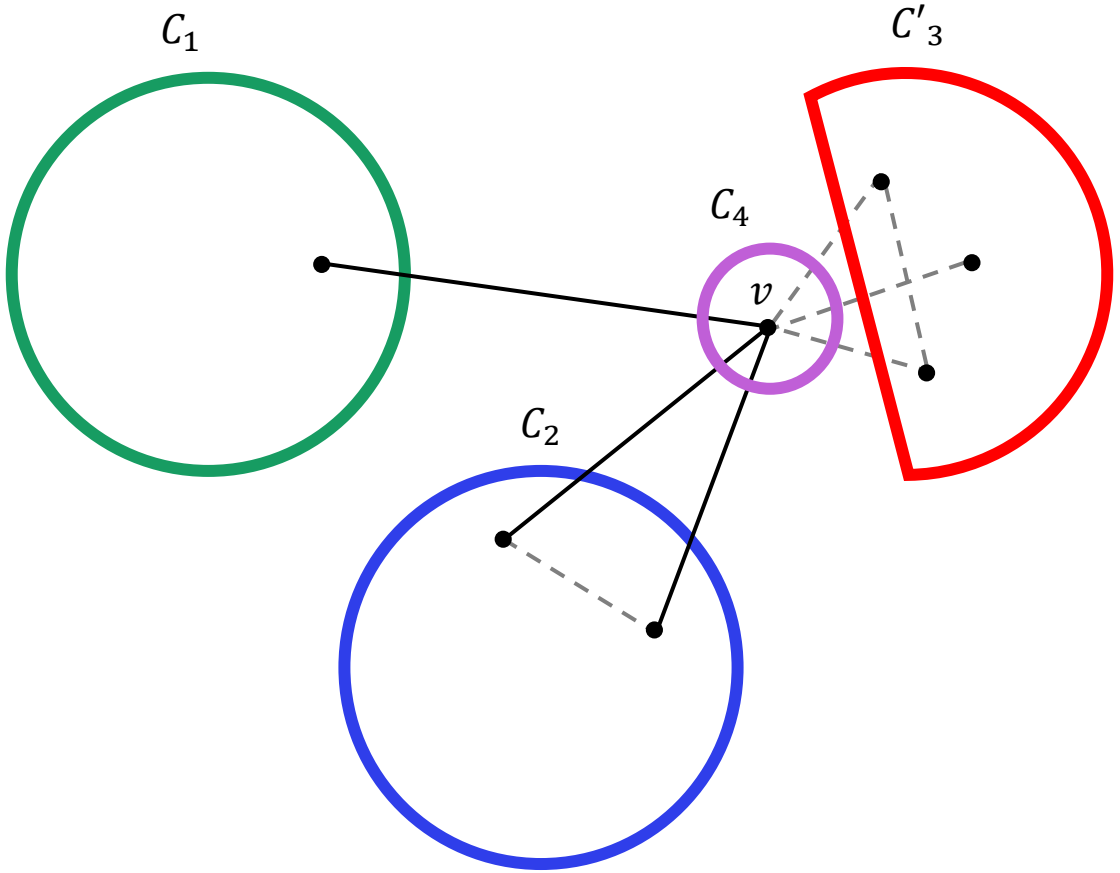
- Remove a vertex v from its community (here C_3). → Modularity change
- Put v into its own community (here C_4). → Modularity change



Louvain algorithm: the main idea

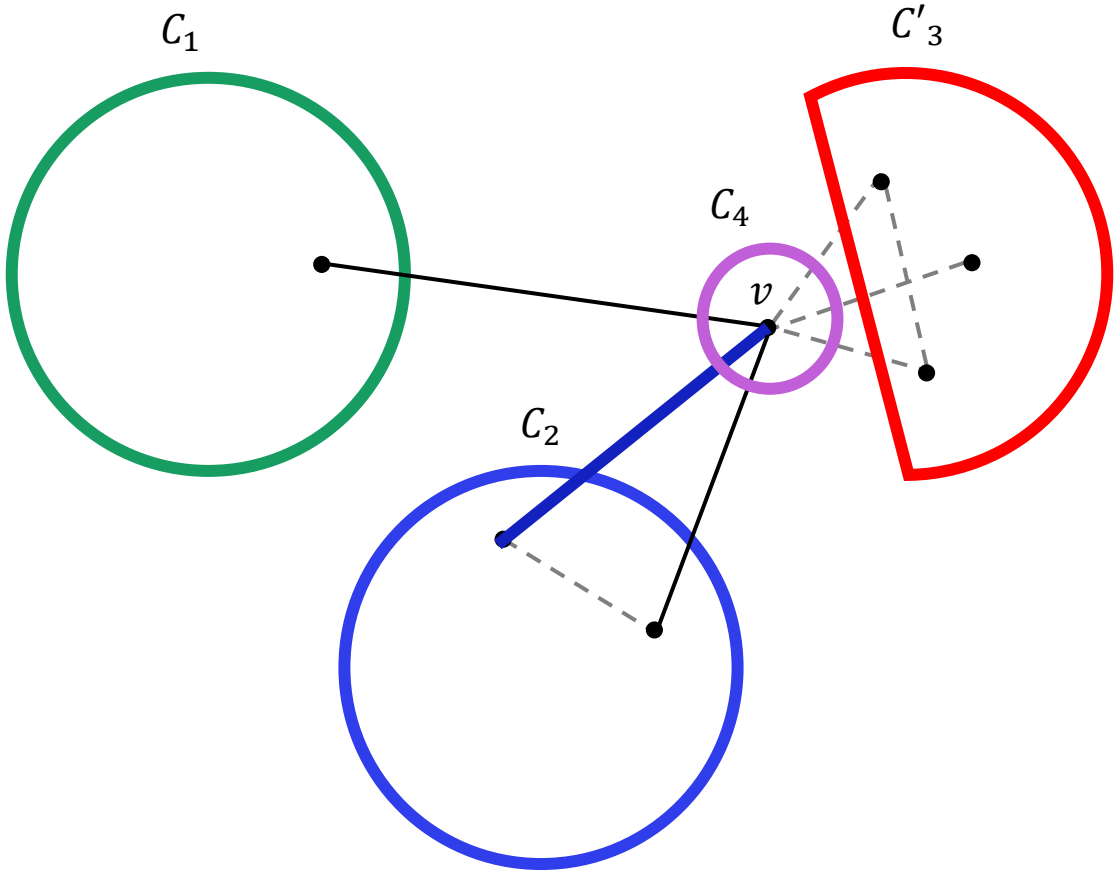
- Modularity change when switching to this refined clustering:

$$\Delta Q_{remove} := Q(G, C_1, C_2, C'_3, C_4) - Q(G, C_1, C_2, C_3)$$



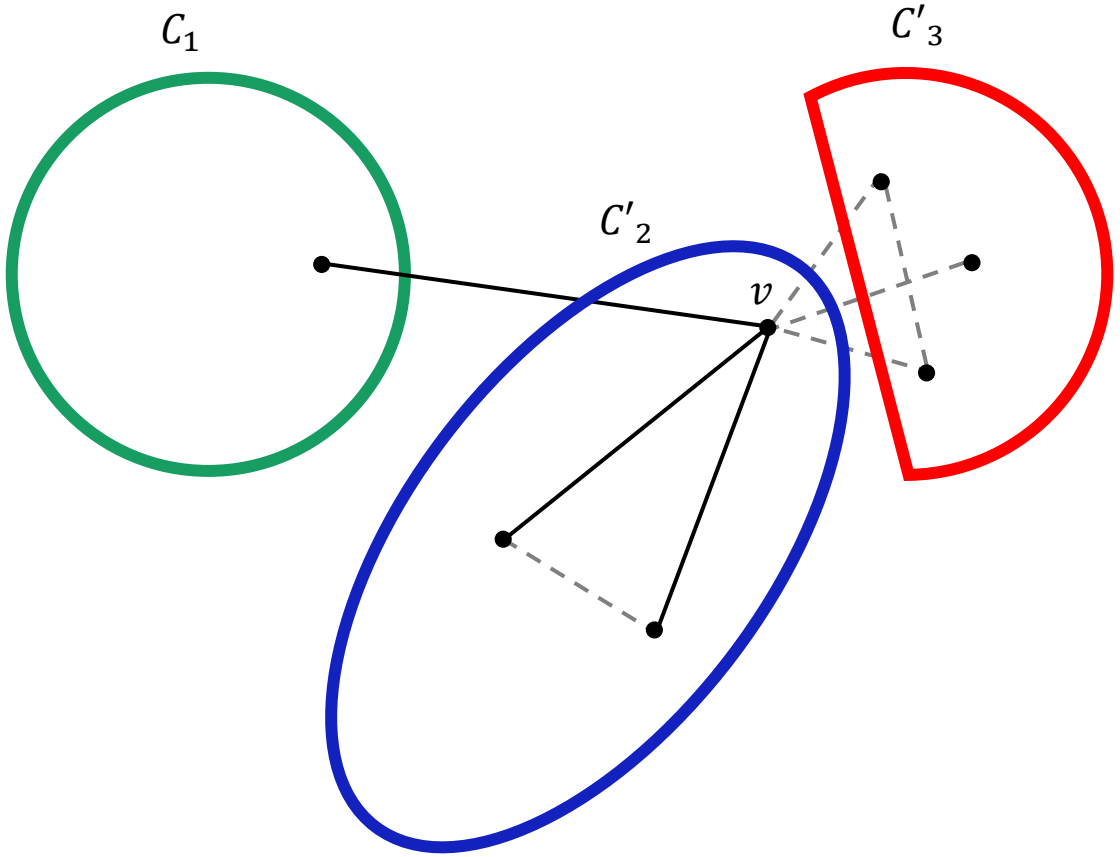
Louvain algorithm: the main idea

- Now move v to a community that contains a neighbour of v.



Louvain algorithm: the main idea

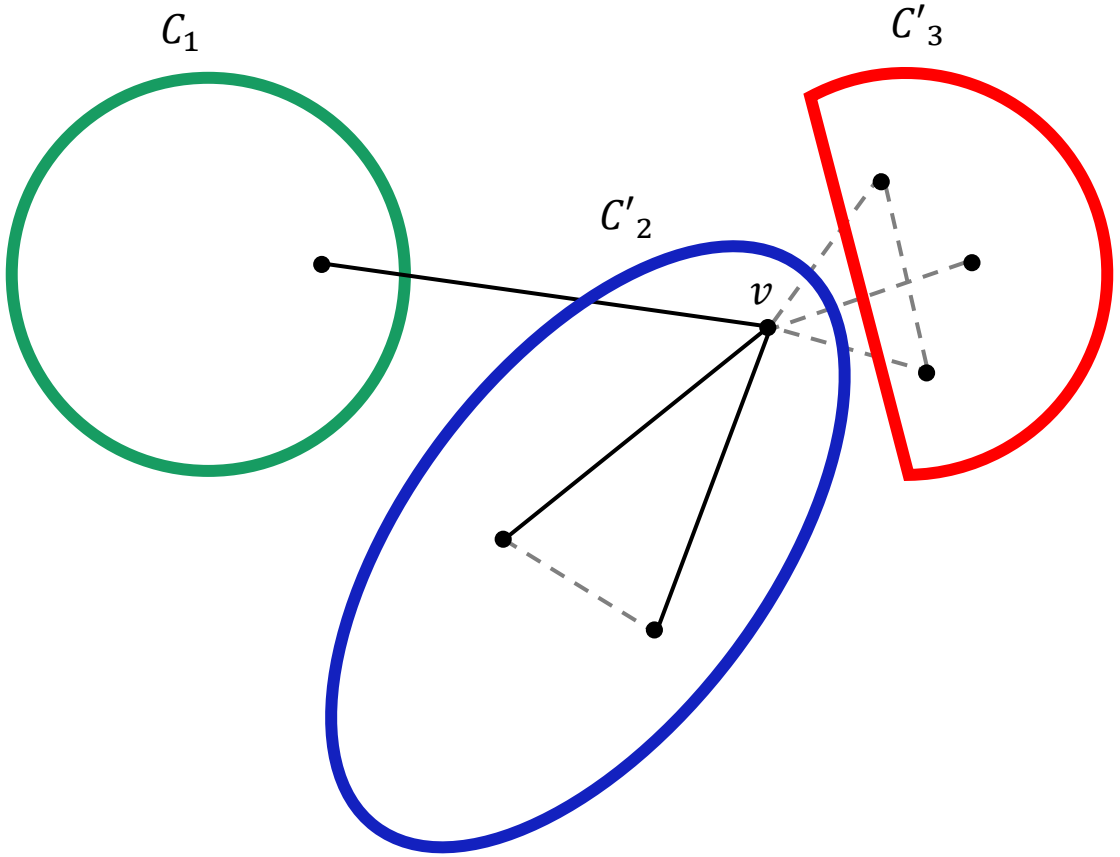
- Now move v to a community that contains a neighbour of v



Louvain algorithm: the main idea

- Modularity change when switching to this clustering:

$$\Delta Q_{insert} := Q(G, C_1, C'_2, C'_3, C_4) - Q(G, C_1, C_2, C'_3, C_4)$$



Louvain algorithm: the main idea

- The total change in modularity is:

$$\Delta Q_{tot} := \Delta Q_{remove} + \Delta Q_{insert}$$

Louvain algorithm: the main idea

- The total change in modularity is:

$$\Delta Q_{tot} := \Delta Q_{remove} + \Delta Q_{insert}$$

- Note that ΔQ_{insert} depends on to which community we move v.

Louvain algorithm: the main idea

- The total change in modularity is:

$$\Delta Q_{tot} := \Delta Q_{remove} + \Delta Q_{insert}$$

- Note that ΔQ_{insert} depends on to which community we move v.

  → Check all and choose community with biggest modularity gain.

Louvain algorithm: the main idea

- The total change in modularity is:

$$\Delta Q_{tot} := \Delta Q_{remove} + \Delta Q_{insert}$$

- Note that ΔQ_{insert} depends on to which community we move v.

  → Check all and choose community with biggest modularity gain.

- ΔQ_{remove} and ΔQ_{insert} can be computed locally by only considering those communities that are affected by the change.
  - (See e.g. Stanford lecture notes from 2021 on Machine Learning with Graphs by Leskovec)

Louvain algorithm: the full picture of step 1

- Start with a vertex partition into singleton sets.

Louvain algorithm: the full picture of step 1

- Start with a vertex partition into singleton sets.
- Order the vertices arbitrarily: $v_1, v_2, \dots, v_n$.

Louvain algorithm: the full picture of step 1

- Start with a vertex partition into singleton sets.
- Order the vertices arbitrarily: $v_1, v_2, \dots, v_n$.
- Compute ΔQ_{tot} when moving vertex v_1.

Louvain algorithm: the full picture of step 1

- Start with a vertex partition into singleton sets.
- Order the vertices arbitrarily: $v_1, v_2, \ldots, v_n$.
- Compute ΔQ_{tot} when moving vertex v_1.
- **IF** $\Delta Q_{tot} \leq 0$, do not change the community where v_1 is in.

Louvain algorithm: the full picture of step 1

- Start with a vertex partition into singleton sets.
- Order the vertices arbitrarily: $v_1, v_2, \ldots, v_n$.
- Compute ΔQ_{tot} when moving vertex v_1.
- **IF** $\Delta Q_{tot} \leq 0$, do not change the community where v_1 is in.
- Otherwise, change the communities so that ΔQ_{tot} is maximal.

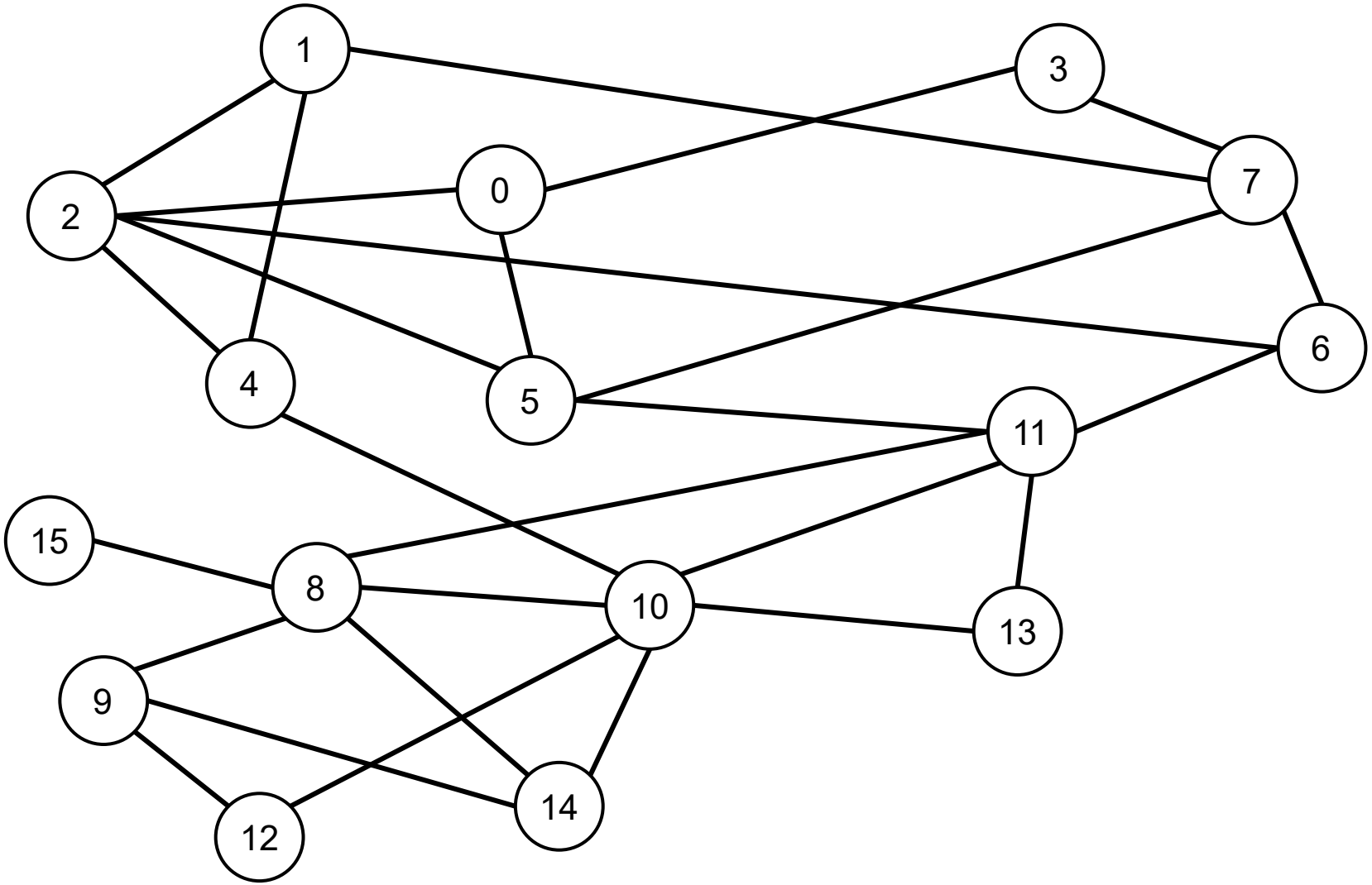
Louvain algorithm: the full picture of step 1

- Start with a vertex partition into singleton sets.
- Order the vertices arbitrarily: $v_1, v_2, \dots, v_n$.
- Compute ΔQ_{tot} when moving vertex v_1.
- **IF** $\Delta Q_{tot} \leq 0$, do not change the community where v_1 is in.
- Otherwise, change the communities so that ΔQ_{tot} is maximal.
- Continue with v_2 in the same way. (The only difference is that one community consists now of 2 vertices instead of 1.)

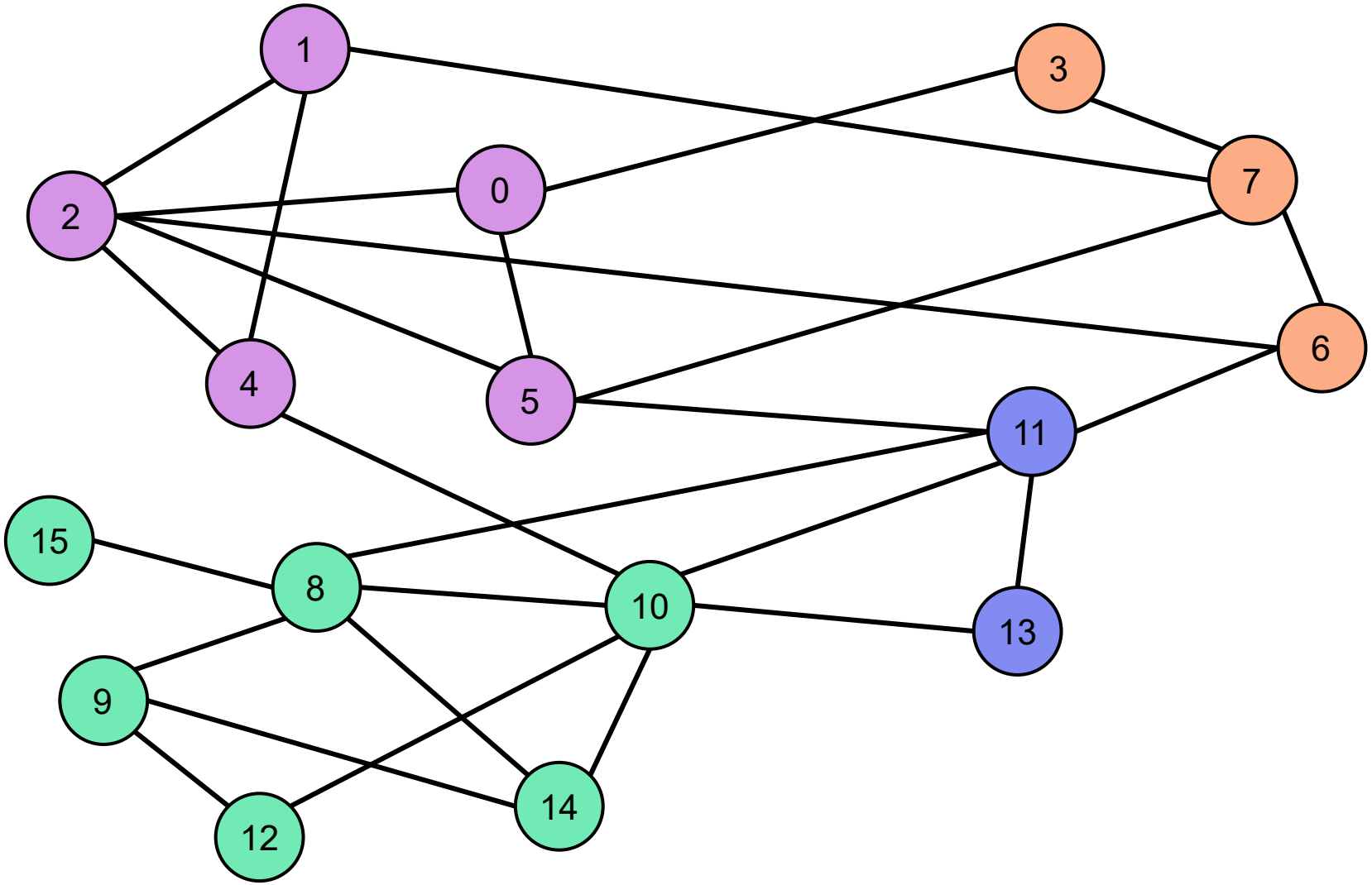
Louvain algorithm: the full picture of step 1

- Start with a vertex partition into singleton sets.
- Order the vertices arbitrarily: $v_1, v_2, \ldots, v_n$.
- Compute ΔQ_{tot} when moving vertex v_1.
- **IF** $\Delta Q_{tot} \leq 0$, do not change the community where v_1 is in.
- Otherwise, change the communities so that ΔQ_{tot} is maximal.
- Continue with v_2 in the same way. (The only difference is that one community consists now of 2 vertices instead of 1.)
- Stop if all vertices have been processed.

Louvain algorithm: illustration



Louvain algorithm: end of step 1



Louvain algorithm: step 2

- Collapse each community into a new ”super” vertex.

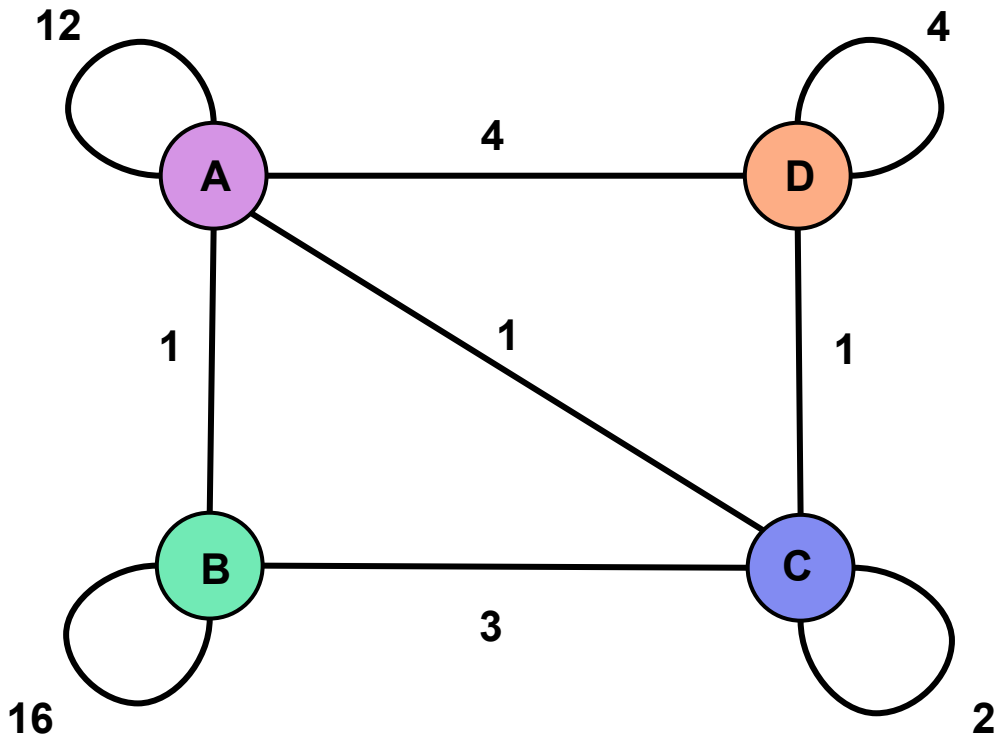
Louvain algorithm: step 2

- Collapse each community into a new ”super” vertex.
- Add loops at each vertex, weighted by 2 times the sum of all (weighted) edges within the former community.

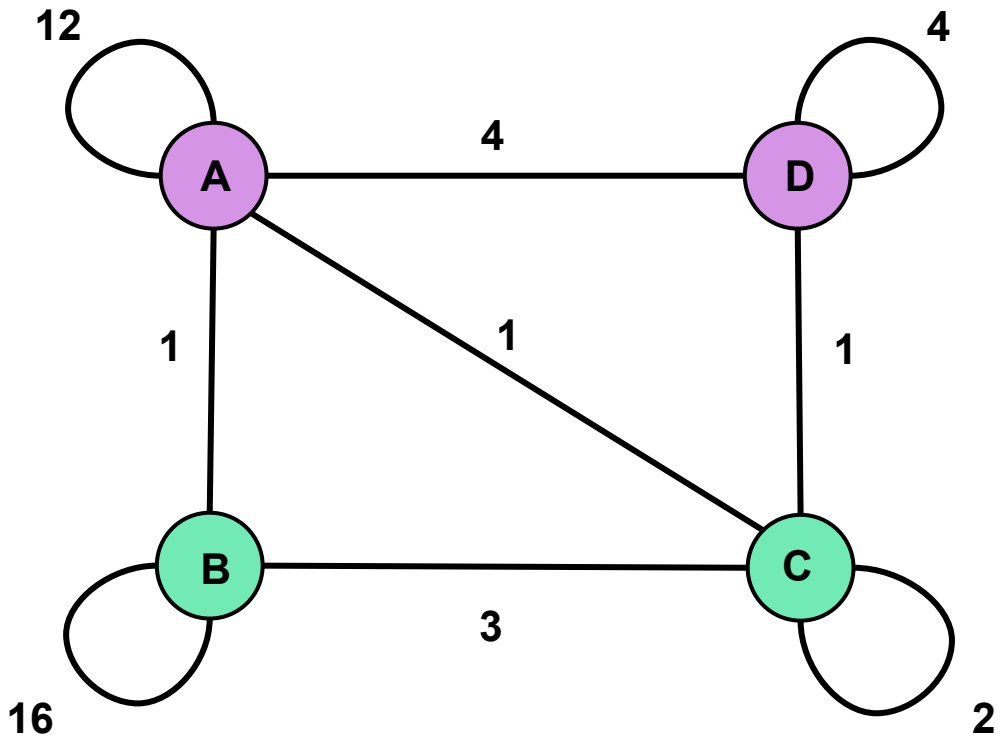
Louvain algorithm: step 2

- Collapse each community into a new ”super” vertex.
- Add loops at each vertex, weighted by 2 times the sum of all (weighted) edges within the former community.
- Replace all edges between two former communities by one edge weighted by the sum of all (weights of) removed edges.

Louvain algorithm: step 2



Louvain algorithm: second pass of step 1



Louvain algorithm: weighted modularity involved

- Recall: Definition of modularity:

$$Q(G,\mathcal{C}) = \frac{1}{2m} \cdot \sum_{C_i \in \mathcal{C}} \sum_{\substack{v,w \in V(C_i) \\ v \neq w}} \left[A_{vw} - \frac{d_G(v) \cdot d_G(w)}{2m} \right]$$

Louvain algorithm: weighted modularity involved

- Recall: Definition of modularity:

$$Q(G,\mathcal{C}) = \frac{1}{2m} \cdot \sum_{C_i \in \mathcal{C}} \sum_{\substack{v,w \in V(C_i) \\ v \neq w}} \left[A_{vw} - \frac{d_G(v) \cdot d_G(w)}{2m} \right]$$

now: sum over all edge weights

Louvain algorithm: weighted modularity involved

- Recall: Definition of modularity:

$$Q(G,\mathcal{C}) = \frac{1}{2m} \cdot \sum_{C_i \in \mathcal{C}} \sum_{\substack{v,w \in V(C_i) \\ v \neq w}} \left[A_{vw} - \frac{d_G(v) \cdot d_G(w)}{2m} \right]$$

now: indicates edge weights

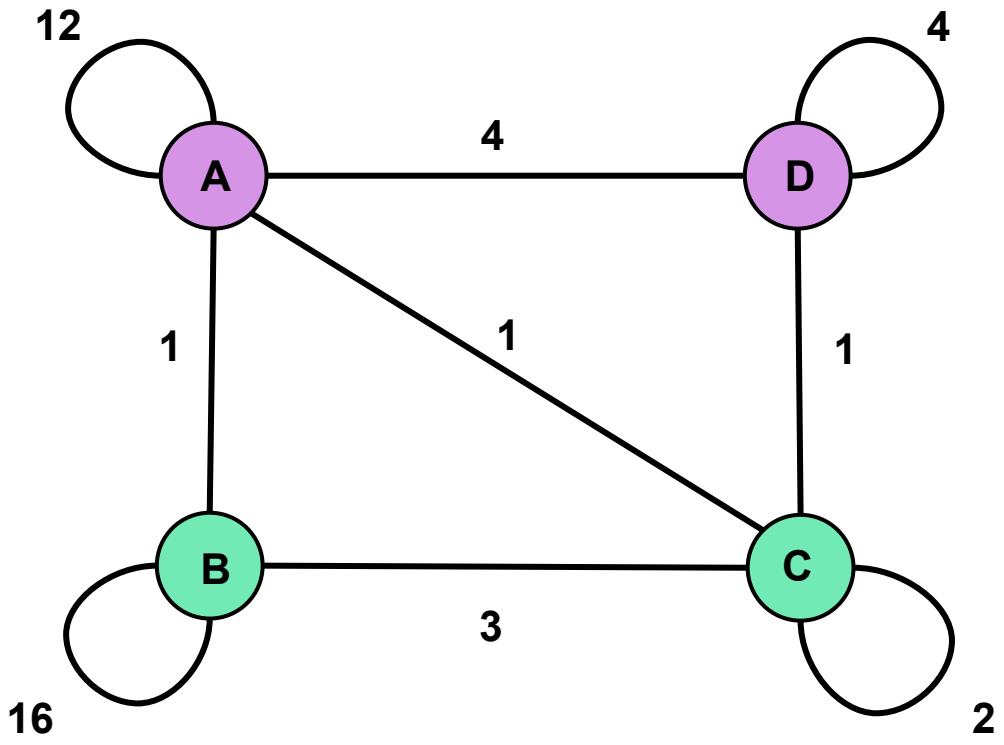
Louvain algorithm: weighted modularity involved

- Recall: Definition of modularity:

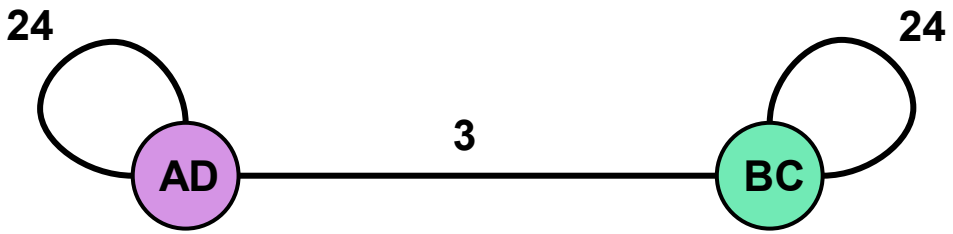
$$Q(G,\mathcal{C}) = \frac{1}{2m} \cdot \sum_{C_i \in \mathcal{C}} \sum_{\substack{v,w \in V(C_i) \\ v \neq w}} \left[A_{vw} - \frac{d_G(v) \cdot d_G(w)}{2m} \right]$$

now: indicate weighted degrees

Louvain algorithm: second pass of step 1



Louvain algorithm: second pass of step 2



Spectral clustering

Remarks about spectral clustering

- Spectral clustering can be applied to several data sets, but you need to **turn** the **data into a graph** (there are several ways).

Remarks about spectral clustering

- Spectral clustering can be applied to several data sets, but you need to **turn** the **data into a graph** (there are several ways).
- For spectral clustering the **data** set does **not have** to be of a **special shape** (e.g. not ‘sperical’ as for k-means). The algorithm adapts to the shape of the data.

Remarks about spectral clustering

- Spectral clustering can be applied to several data sets, but you need to **turn** the **data into a graph** (there are several ways).
- For spectral clustering the **data** set does **not have** to be of a **special shape** (e.g. not ‘sperical’ as for k-means). The algorithm adapts to the shape of the data.
- Spectral clustering yields very good results.

Remarks about spectral clustering

- Spectral clustering can be applied to several data sets, but you need to **turn** the **data into a graph** (there are several ways).
- For spectral clustering the **data** set does **not have** to be of a **special shape** (e.g. not ‘sperical’ as for k-means). The algorithm adapts to the shape of the data.
- Spectral clustering yields very good results.
- **Computing all eigenvalues** is **expensive** for **dense matrices**.
  – But there are quite efficient algorithms for computing only the first few smallest eigenvalues and their eigenvectors.

Spectral clustering

- Let G be a simple undirected graph.
- **Task: Partition $V(G)$ into two classes** (clusters) such that:
  1. we maximise the # of edges within the clusters, and
  2. we minimise the # of edges between the clusters.

Spectral clustering

- Let G be a simple undirected graph.
- **Task: Partition** $V(G)$ **into two classes** (clusters) such that:
  1. we maximise the # of edges within the clusters, and
  2. we minimise the # of edges between the clusters.
- For two vertex sets $A, B \subseteq V(G)$ let $\boldsymbol{E(A,B)}$ denote the set of edges that have one end vertex in A and the other in B.

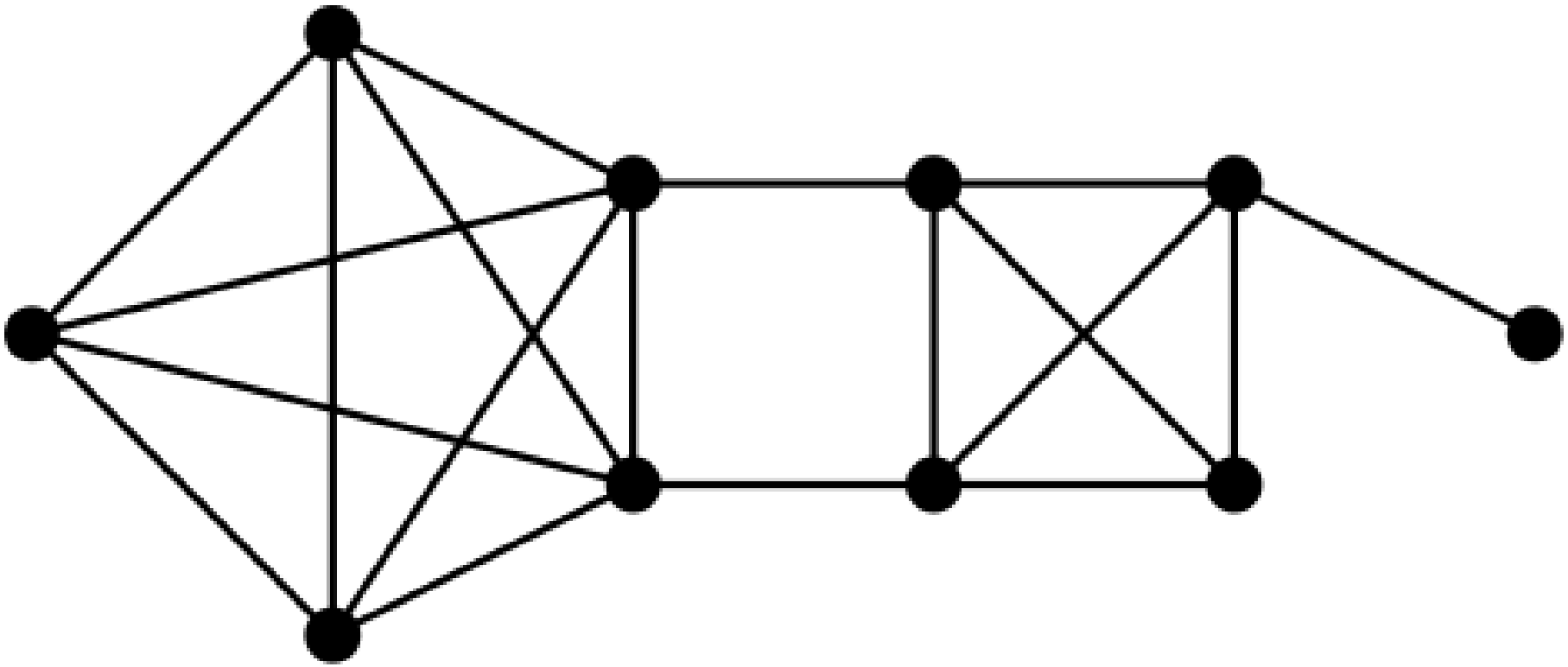
Spectral clustering

- Let G be a simple undirected graph.
- **Task: Partition $V(G)$ into two classes** (clusters) such that:
  1. we maximise the # of edges within the clusters, and
  2. we minimise the # of edges between the clusters.
- For two vertex sets $A, B \subseteq V(G)$ let $\boldsymbol{E(A,B)}$ denote the set of edges that have one end vertex in A and the other in B.
- If $A \subseteq V(G)$ and $B = V(G) \setminus A$ (the complement of A within $V(G)$), then $E(A,B)$ is called a **cut of $\boldsymbol{G}$**.
  (Sometimes we also refer to the partition (A,B) as the cut.)

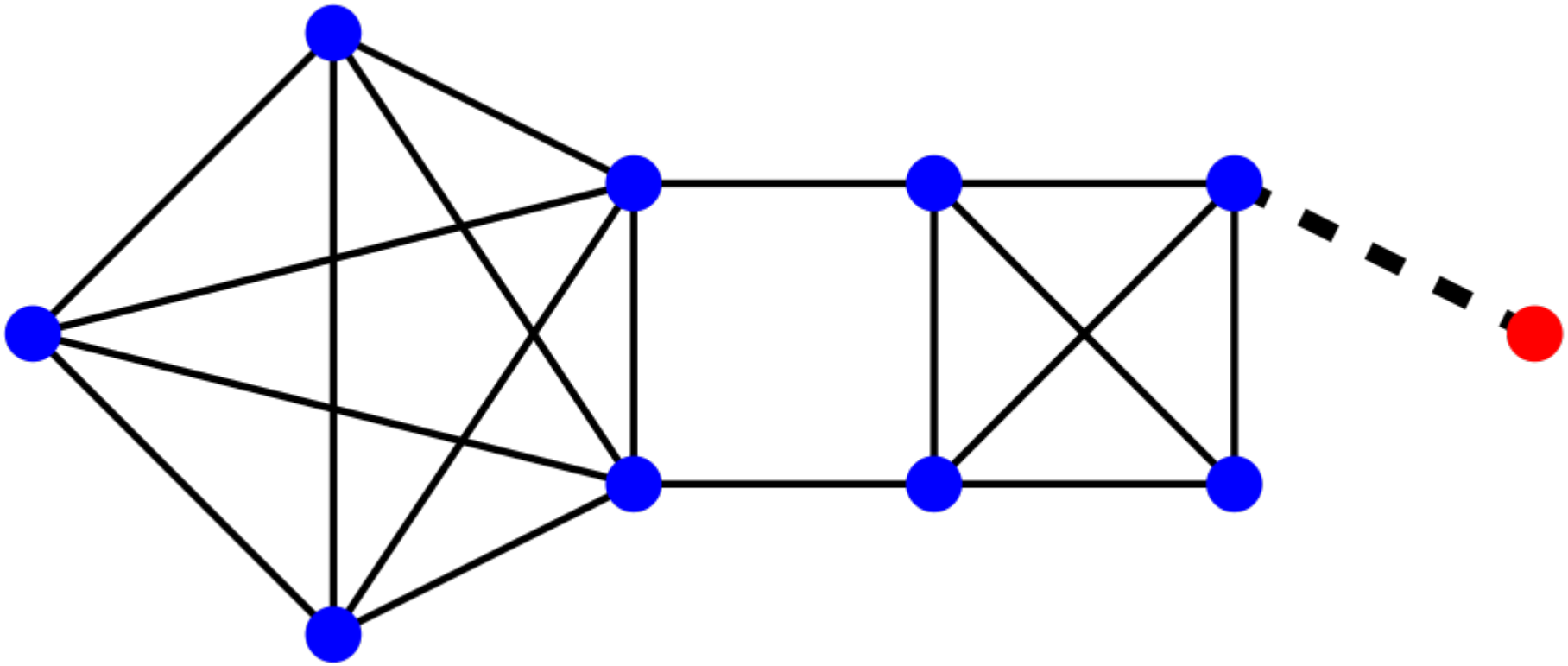
Spectral clustering: “good” cut

- **Idea:** Find the smallest cut $E(A, B)$ within G.
- **Pro:** Can be done (rather) efficiently (Ford-Fulkerson, Karger).
- **Con:** Only focusses on minimising edges between the 2 clusters.
  - What about maximising edges within clusters?

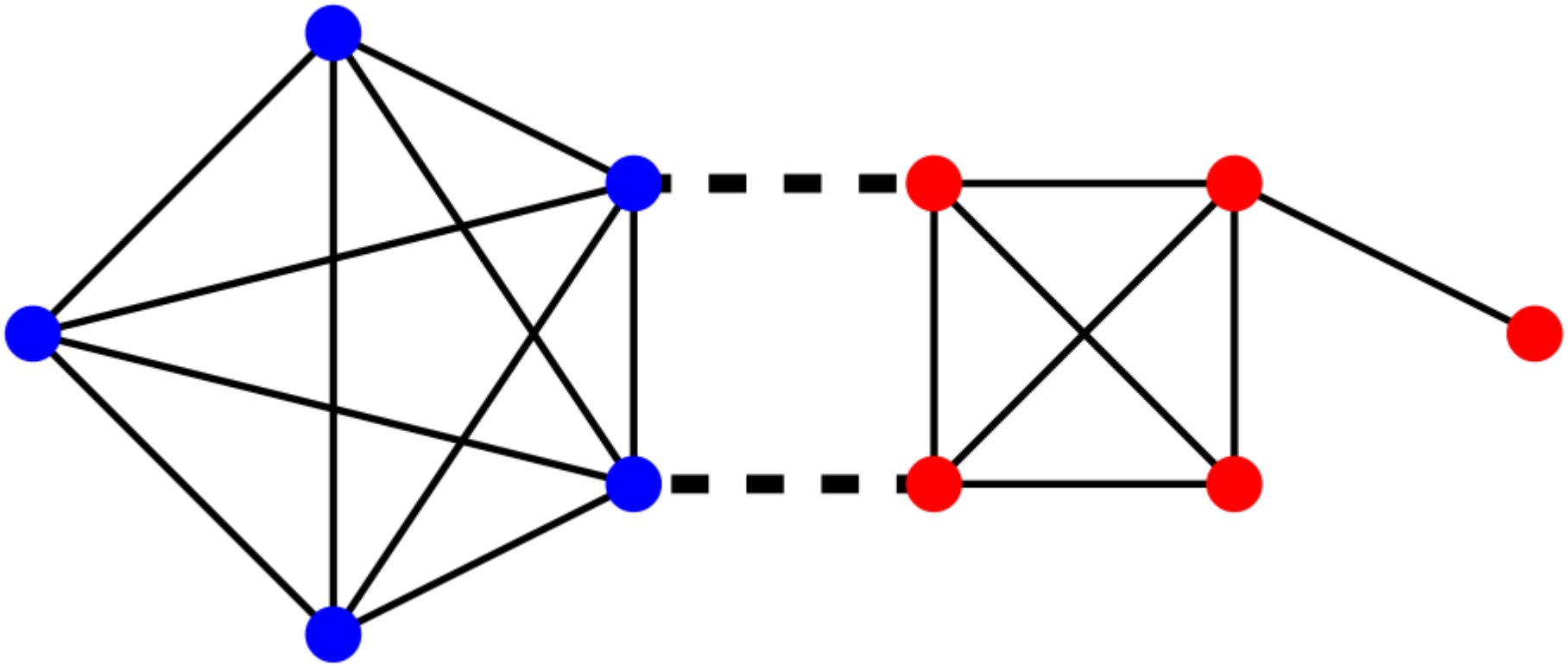
Spectral clustering: “good” cut



Spectral clustering: “good” cut



Spectral clustering: “good” cut



Spectral clustering: “good” cut

- **Better:** Consider ***normalised cuts*** and **minimise**.
- Let $E(A,B)$ be a cut of G. Then the ***normalised cut value*** is:

$$ncut(A,B) := \frac{|E(A,B)|}{vol(A)} + \frac{|E(A,B)|}{vol(B)}$$

Here $\boldsymbol{vol(A)} := \sum_{v \in A} d(v)$.

Spectral clustering: “good” cut

- **Better:** Consider ***normalised cuts*** and **minimise**.
- Let $E(A,B)$ be a cut of G. Then the ***normalised cut value*** is:

$$ncut(A,B) := \frac{|E(A,B)|}{vol(A)} + \frac{|E(A,B)|}{vol(B)}$$

Here $\boldsymbol{vol(A)} := \sum_{v \in A} d(v)$.

- This way we produce more balanced bipartitions, still inducing a small cut.
- However, finding a cut with minimised $ncut$-value is NP-hard!

Spectral clustering: “good” cut

- Let $E(A,B)$ be a cut of G. Then the ***normalised cut value*** is:

$$ncut(A,B) := \frac{|E(A,B)|}{vol(A)} + \frac{|E(A,B)|}{vol(B)}$$

Here $\boldsymbol{vol(A)} := \sum_{v \in A} d(v)$.

- **Or consider: RatioCut**

$$RatioCut(A,B) := \frac{|E(A,B)|}{|A|} + \frac{|E(A,B)|}{|B|}$$

RECAP: Adjacency matrix of a graph

- Let G be a simple undirected graph with $|V(G)| = n$.
  Enumerate the vertices of G by $v_1, \dots, v_n$.

- Define ***Adjacency Matrix*** $A = A(G) \in \mathbb{R}^{n \times n}$ of G as follows:

  Let a_{ij} be the entry of A in row i and column j.

  Set $a_{ij} = 1$ iff $v_i v_j \in E(G)$, otherwise set $a_{ij} = 0$.

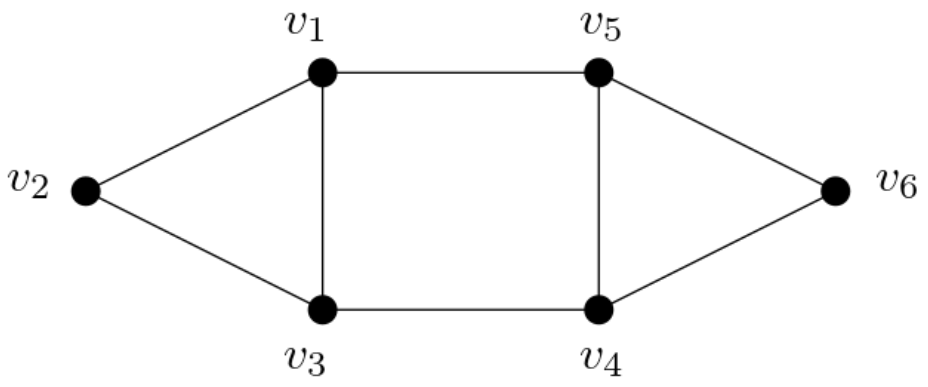
- If $Ax = y$, then the following holds for the i-th component y_i of y:

$$y_i = \sum_{j=1}^{n} a_{ij}\, x_j = \sum_{v_i v_j \in E(G)} x_j$$

RECAP: Adjacency matrix of a graph

Example:

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$



Spectral Graph Theory

- Analysing the ***spectrum*** of a graph G, i.e. of its adjacency matrix A to obtain insight about the structure of G.

- The **spectrum** of A is the set $\boldsymbol{\Lambda} = \{\lambda_1, \dots, \lambda_n\} \subseteq \mathbb{R}$ of its **eigenvalues**. Usually we sort Λ by $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$.

- **Recap** (**eigenvalue** λ and one **eigenvector** x of it):

$$Ax = \lambda \cdot x$$

RECAP: symmetric matrices

- $\boldsymbol{A}(\boldsymbol{G}) = \boldsymbol{A}$ is a **symmetric real matrix**. Hence:
  1. A is diagonalisable.
  2. There is an orthogonal basis of eigenvectors.
  3. A has only real eigenvalues.

- Also: $A(G) = A$ is ***positive semidefinite*** (i.e. all eigenvalues ≥ 0).

- Fact: A is positive semidefinite iff $x^T A x \geq 0$ for all $x \neq 0$.

Some intuition: d-regular graphs

- As special case: Let G be a **connected** graph where each vertex has the same degree $d \in \mathbb{N}$ (the latter property is called $\boldsymbol{d}$**-regular**).

Some intuition: d-regular graphs

- As special case: Let G be a **connected** graph where each vertex has the same degree $d \in \mathbb{N}$ (the latter property is called $\boldsymbol{d}$**-regular**).

$$A \cdot (1, 1, \dots, 1)^T = d \cdot (1, 1, \dots, 1)^T$$

- Why? Because for each vertex, we sum over all its d neighbours.

Some intuition: d-regular graphs

- As special case: Let G be a **connected** graph where each vertex has the same degree $d \in \mathbb{N}$ (the latter property is called $\boldsymbol{d}$**-regular**).

$$A \cdot (1, 1, \dots, 1)^T = d \cdot (1, 1, \dots, 1)^T$$

- Why? Because for each vertex, we sum over all its d neighbours.

Actually:

1. d is the largest eigenvalue of A.
2. d has multiplicity 1. So there is only one eigenvector for it.

Some intuition: d-regular graphs

- As special case: Let G be d-regular with precisely **2 components**, say X and Y.

Some intuition: d-regular graphs

- As special case: Let G be d-regular with precisely **2 components**, say X and Y.

$$A \cdot (\underbrace{1, \dots, 1}_{X}, \underbrace{0, \dots, 0}_{Y})^T = d \cdot (\underbrace{1, \dots, 1}_{X}, \underbrace{0, \dots, 0}_{Y})^T$$

Some intuition: d-regular graphs

- As special case: Let G be d-regular with precisely **2 components**, say X and Y.

$$A \cdot (\underbrace{1, \dots, 1}_{X}, \underbrace{0, \dots, 0}_{Y})^T = d \cdot (\underbrace{1, \dots, 1}_{X}, \underbrace{0, \dots, 0}_{Y})^T$$

$$A \cdot (\underbrace{0, \dots, 0}_{X}, \underbrace{1, \dots, 1}_{Y})^T = d \cdot (\underbrace{0, \dots, 0}_{X}, \underbrace{1, \dots, 1}_{Y})^T$$

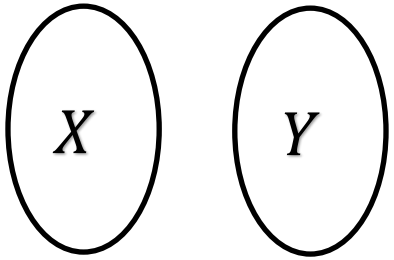
Some intuition: d-regular graphs

- As special case: Let G be d-regular with precisely **2 components**, say X and Y.

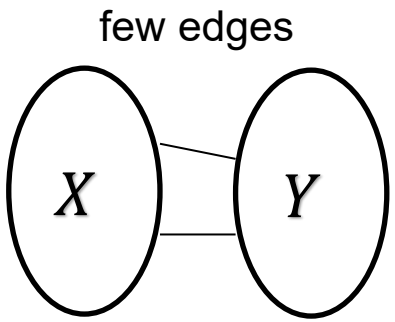
$$A \cdot (\underbrace{1, \dots, 1}_{X}, \underbrace{0, \dots, 0}_{Y})^T = d \cdot (\underbrace{1, \dots, 1}_{X}, \underbrace{0, \dots, 0}_{Y})^T$$

$$A \cdot (\underbrace{0, \dots, 0}_{X}, \underbrace{1, \dots, 1}_{Y})^T = d \cdot (\underbrace{0, \dots, 0}_{X}, \underbrace{1, \dots, 1}_{Y})^T$$

- Now: $\lambda_n = \lambda_{n-1} = d$.
- **Intuition:**



$\lambda_n - \lambda_{n-1} = 0$



$\lambda_n - \lambda_{n-1} \approx 0$

General case

- Let G be a simple undirected connected graph.
- Define ***Degree Matrix D*** $= D(G) \in \mathbb{R}^{n \times n}$ of G as follows:

  Let d_{ij} be the entry of D in row i and column j.

  Set $d_{ij} = 0$ for all $i \neq j$ and $d_{ii} = d(v_i)$ for all i.
- Define the **Laplacian Matrix** $\boldsymbol{L} = L(G) \in \mathbb{R}^{n \times n}$ of G as $A - D$.

Facts about L:

- L is symmetric
- L is diagonalisable.
- There is an orthogonal basis of eigenvectors.
- L has only real eigenvalues and $\lambda_1 = 0$ witnessed by $(1, \dots, 1)^T$.
- L is positive semidefinite, hence $x^T L x \geq 0$ for all $x \neq 0$..

General case

- **Fact:** If $\boldsymbol{x}$ is **orthogonal to** the **eigenvector of** $\boldsymbol{\lambda_1}$ (so: $\sum_i x_i = 0$) and x is **normalised**, i.e. $x^T x = 1$, then the following holds (by Rayleigh):

$$\lambda_2 = \min_x x^T L x = \sum_{v_i v_j \in E(G)} \left(x_i - x_j\right)^2$$

General case

- **Fact:** If $\boldsymbol{x}$ is **orthogonal to** the **eigenvector of** $\boldsymbol{\lambda_1}$ (so: $\sum_i x_i = 0$) and x is **normalised**, i.e. $x^T x = 1$, then the following holds (by Rayleigh):

$$\lambda_2 = \min_x x^T L x = \sum_{v_i v_j \in E(G)} \left(x_i - x_j\right)^2$$

- Furthermore, a minimising vector x is an eigenvector for λ_2.

General case

- **Fact:** If $\boldsymbol{x}$ is **orthogonal to** the **eigenvector of** $\boldsymbol{\lambda_1}$ (so: $\sum_i x_i = 0$) and x is **normalised**, i.e. $x^T x = 1$, then the following holds (by Rayleigh):

$$\lambda_2 = \min_x x^T L x = \sum_{v_i v_j \in E(G)} \left(x_i - x_j\right)^2$$

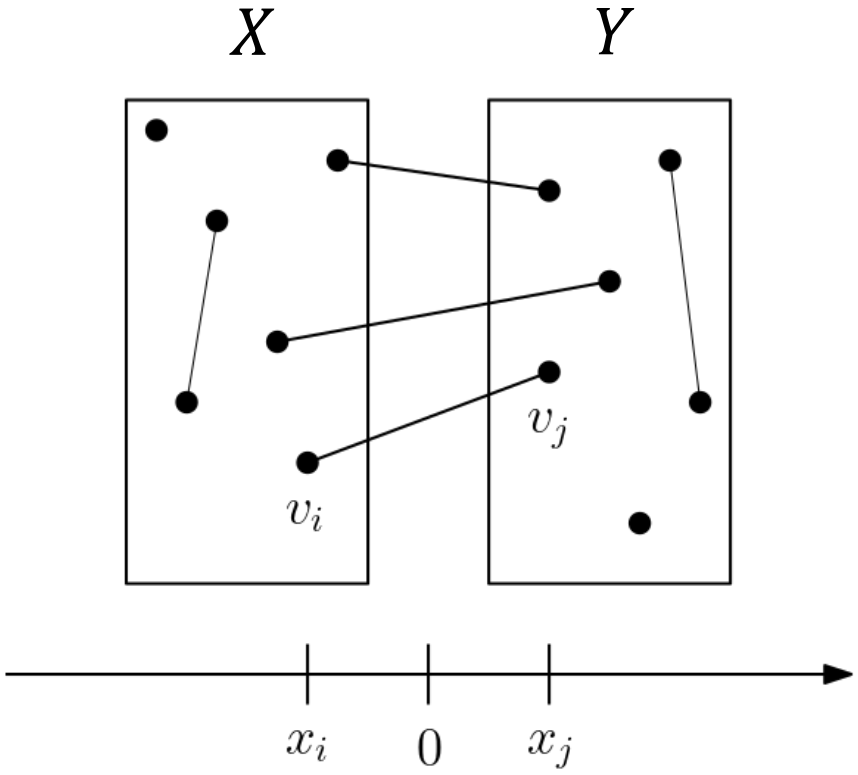
- Furthermore, a minimising vector x is an eigenvector for λ_2.

- **CLUSTERING:** Put all v_i with $x_i < 0$ into set X, the rest into set Y. This yields bipartition of $V(G)$.

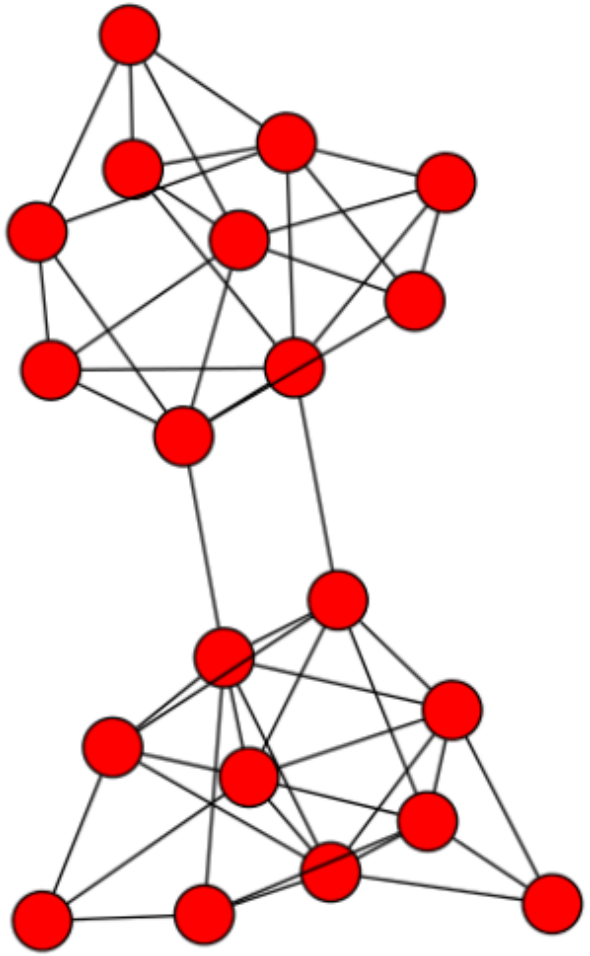
General case

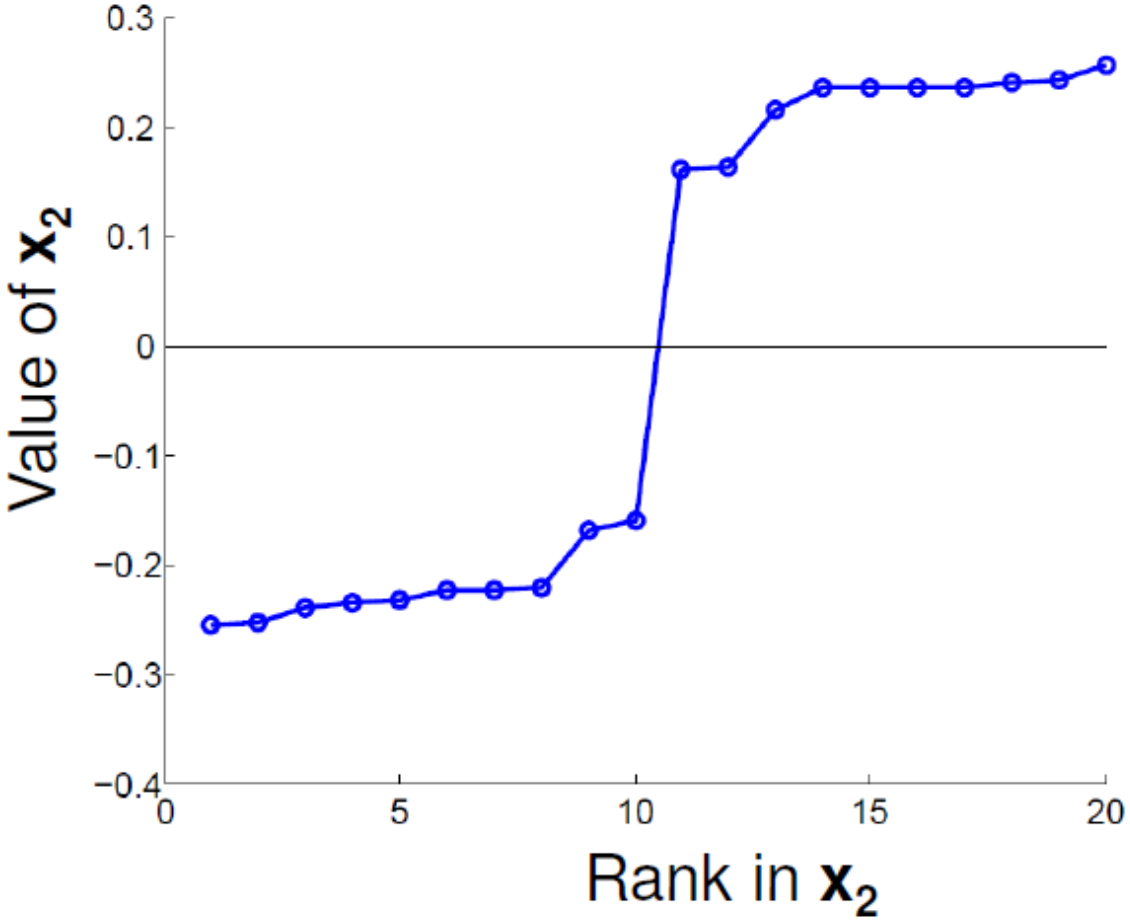
$$\lambda_2 = \min_{\substack{x \\ \sum_i x_i = 0}} x^T L x = \sum_{v_i v_j \in E(G)} \left(x_i - x_j\right)^2$$

- If $v_i v_j \in E(G)$, ideally $x_i \approx x_j$.
- Indication for $|X| \approx |Y|$.
- But some x_i are > 0 and some < 0, since $x \neq 0$ and $\sum_i x_i = 0$.

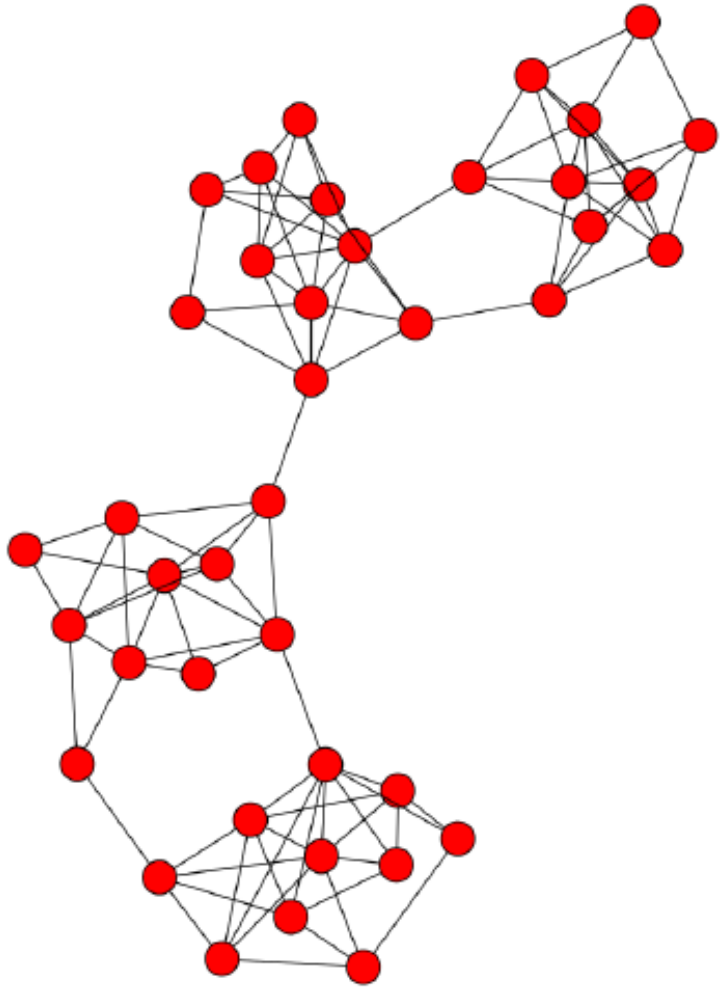


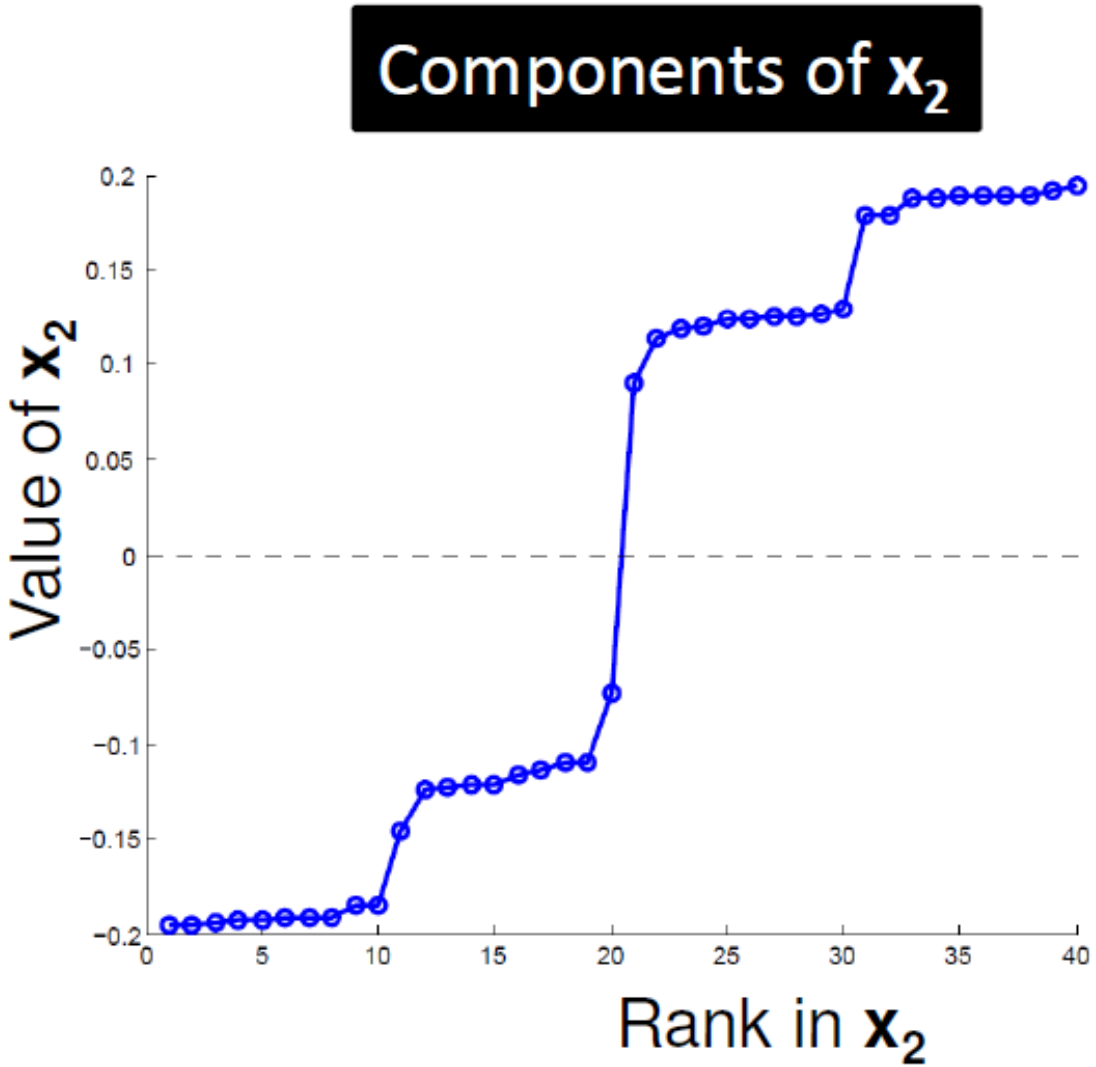
Example 1




0.3
0.2
0.1
0
−0.1
−0.2
−0.3
−0.4
0
5
10
15
20
Value of $\mathbf{x}_2$
Rank in $\mathbf{x}_2$

Example 2




Components of $\mathbf{x}_2$
Value of $\mathbf{x}_2$
Rank in $\mathbf{x}_2$
0.2
0.15
0.1
0.05
0
−0.05
−0.1
−0.15
−0.2
0
5
10
15
20
25
30
35
40

Indication for good balanced partition

- Let $E(X,Y)$ be a cut of G. Then define:

$$\alpha = \alpha(X,Y) = \frac{|E(X,Y)|}{min\{|X|,|Y|\}}$$

- Let $\boldsymbol{\Delta(G)}$ denote the **maximum degree** of G.
  Then the following (Cheeger inequality) holds:

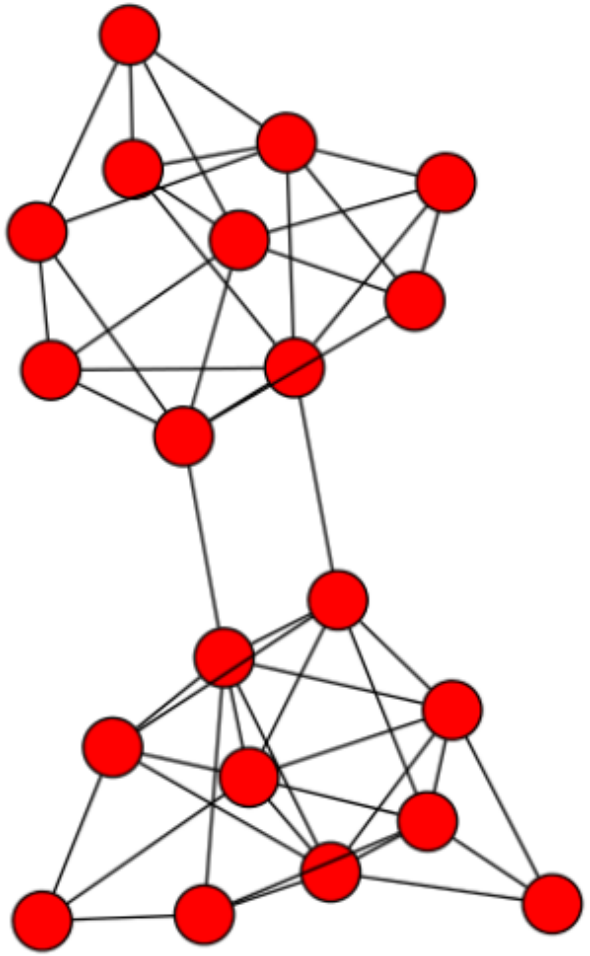
$$\frac{\alpha^2}{2\Delta(G)} \leq \lambda_2 \leq 2\alpha$$

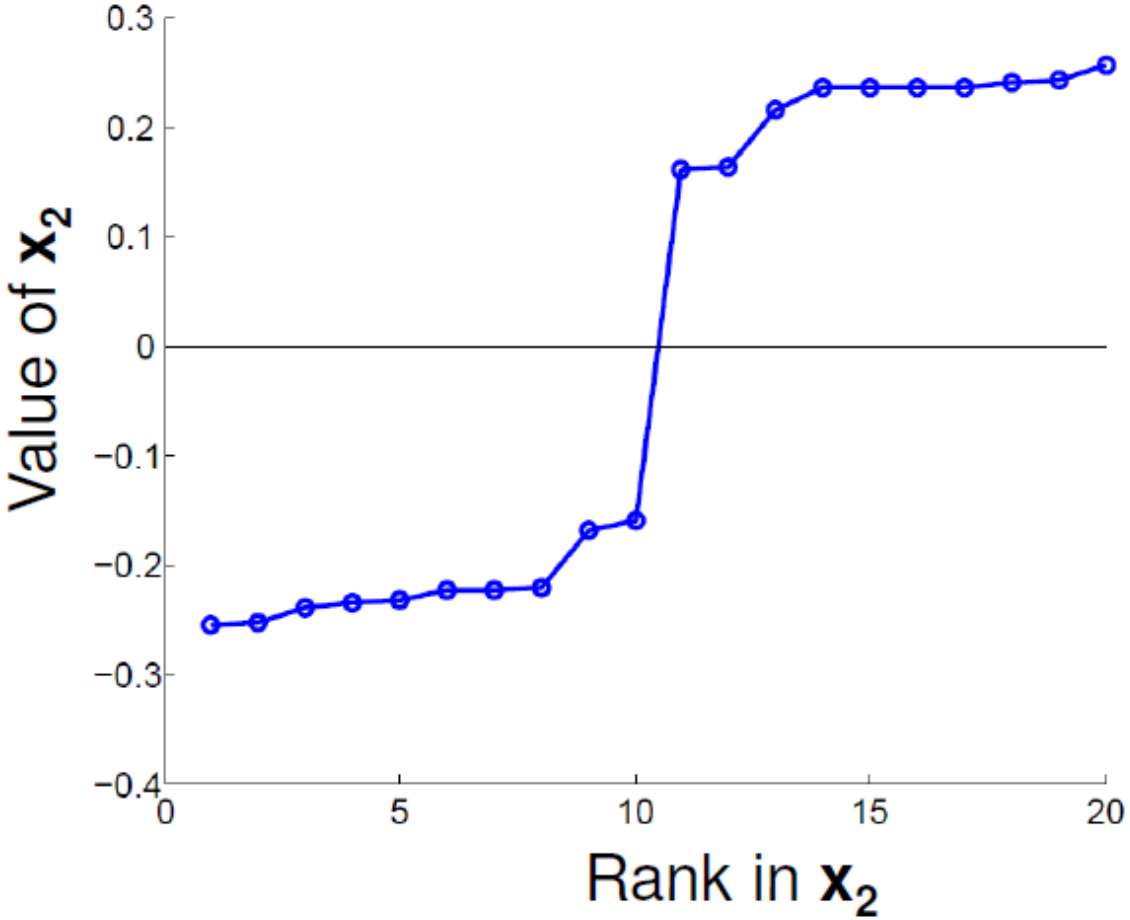
- Hence, we approximately (at most factor 2) find some balanced (w.r.t. α) cut $E(X,Y)$ via an eigenvector of L corresponding to λ_2.

Spectral clustering mixed with k-means

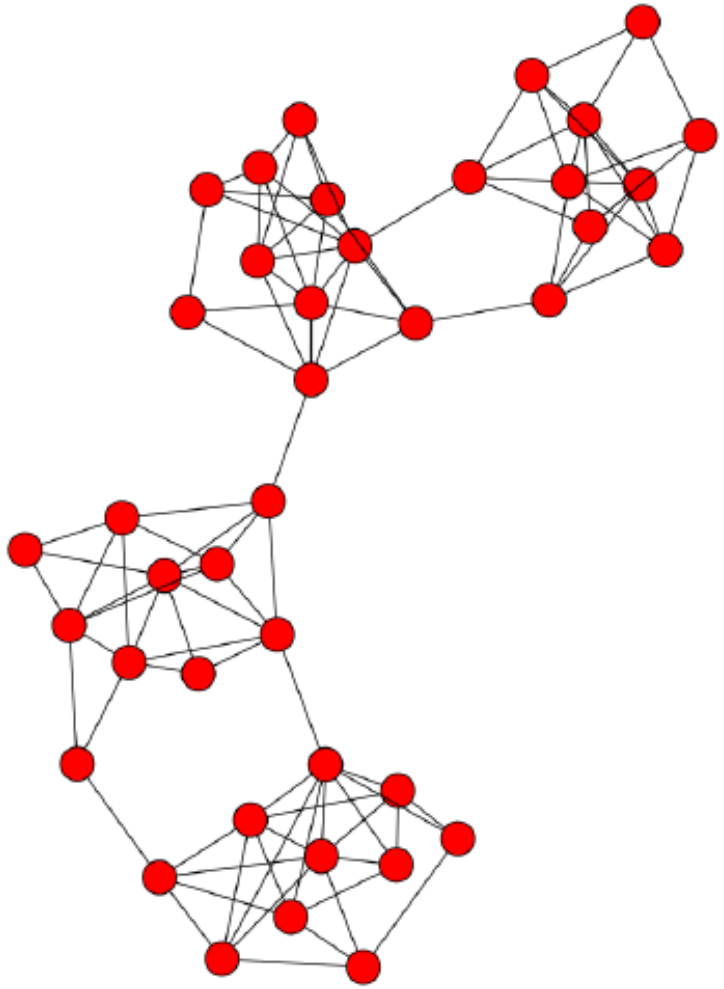
- Often: building the bipartition (X, Y) for $V(G)$ from the components of the eigenvector x_2 of λ_2 by checking the sign is not ideal.
  - Some other threshold than 0 might be better (especially if we partition into more than 2 clusters).
- **Idea:** Perform k-means algorithm on the entries of x_2, e.g. just in $\mathbb{R}^1$.
  - A similar approach works for partitioning into k clusters. There we consider entries of several eigenvectors (w.r.t. further eigenvalues) as vectors of $\mathbb{R}^{k-1}$.

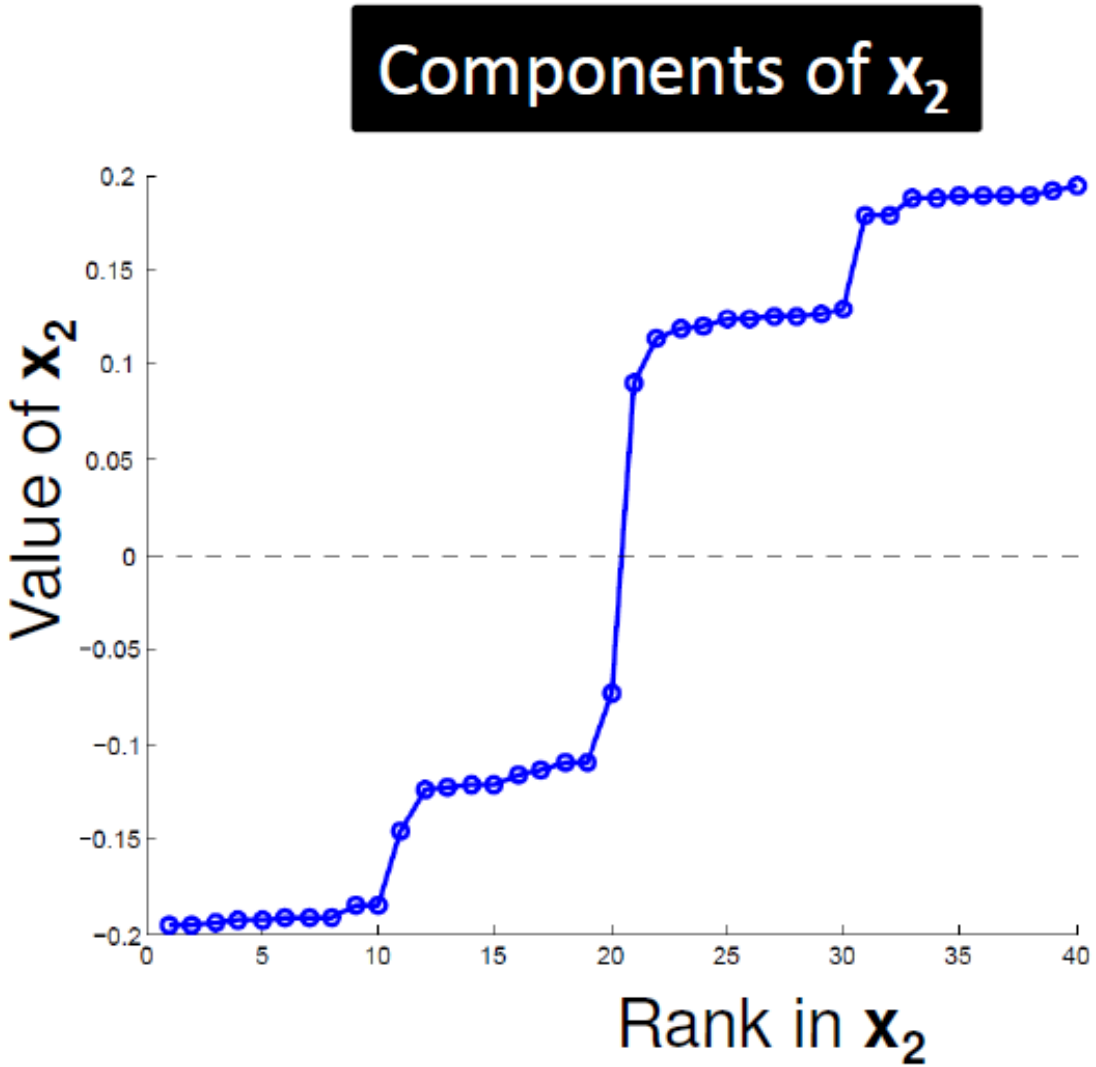
Example 1




0.3
0.2
0.1
0
−0.1
−0.2
−0.3
−0.4
0
5
10
15
20
Value of $\mathbf{x}_2$
Rank in $\mathbf{x}_2$

Example 2




Components of $\mathbf{x}_2$
Value of $\mathbf{x}_2$
Rank in $\mathbf{x}_2$
0.2
0.15
0.1
0.05
0
−0.05
−0.1
−0.15
−0.2
0
5
10
15
20
25
30
35
40

Spectral clustering with k clusters

- **Idea 1 (naive):** Recursively apply bipartitioning algorithm in a divisive hierarchical manner.
  - **Con:** Not very efficient.
- **Idea 2 (better):** Use more eigenvectors, also for bigger eigenvalues. Can be done similarly, but with a normalised Laplacian.
  - Preferable and commonly used.
- Efficient approximation algorithms (up to a constant factor) for k clusters w.r.t similar normalised cut conditions do not exist.