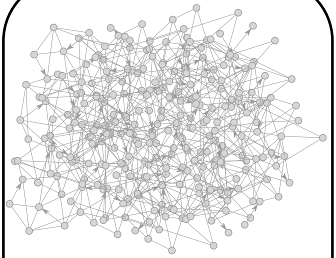
CS 3824: The Louvain and Leiden Algorithms

T. M. Murali

September 27 and 29, 2022

Problem Formulation

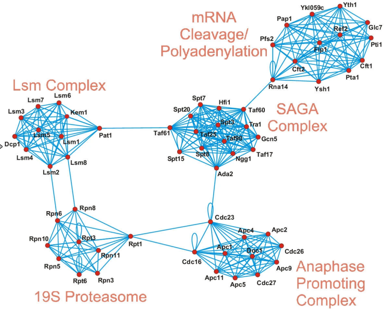
Input



Human Interactome

Module finding algorithm

Output

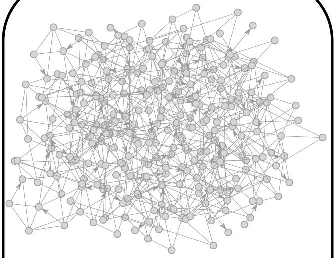


Set of modules

- Given a protein interaction network $G = (V, E, W)$, compute the modules (clusters) in it.

Problem Formulation

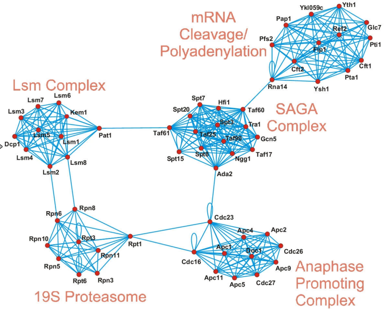
Input



Human Interactome

Module finding algorithm

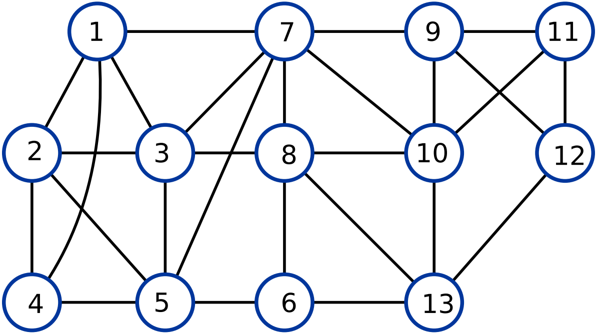
Output



Set of modules

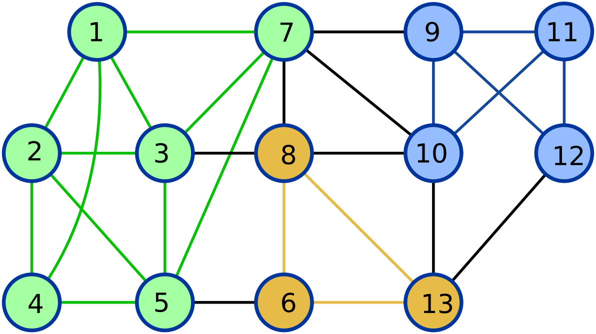
- Given a protein interaction network $G = (V, E, W)$, compute the modules (clusters) in it.
- First, define the quality of a set of modules. Then, develop algorithm to optimize the quality.

Motivation



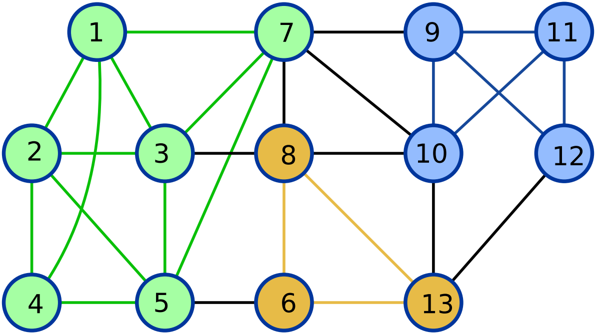
- Given an undirected, unweighted graph $G = (V, E)$ suppose we partition the nodes into k modules $\mathcal{C} = C_1, C_2, \ldots C_k$.
- How do we measure the "quality" of $\mathcal{C}$?
- Intuition: many more edges within modules than among modules.

Motivation



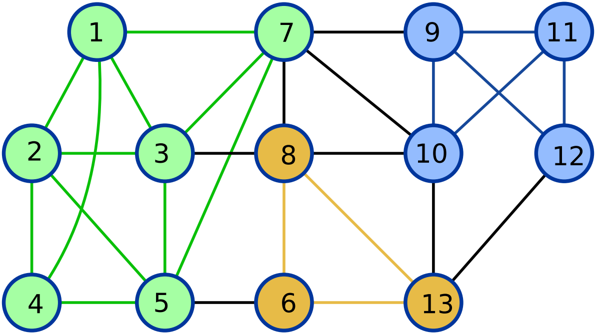
- Given an undirected, unweighted graph $G = (V, E)$ suppose we partition the nodes into k modules $\mathcal{C} = C_1, C_2, \dots C_k$.
- How do we measure the "quality" of $\mathcal{C}$?
- Intuition: many more edges within modules than among modules.

Initial Definition of Modularity



- How do we count the number of edges within modules?

Initial Definition of Modularity

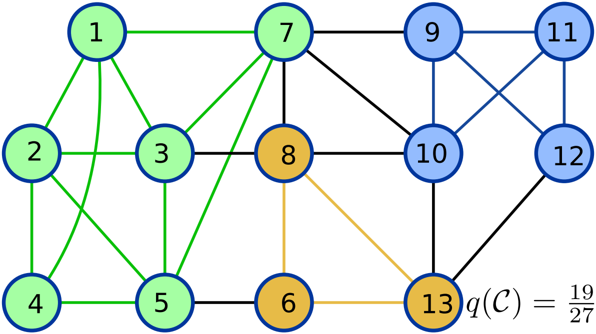


- How do we count the number of edges within modules?
- For every node $u \in V$, define $c(u)$ as the index of u's module.

$$q(\mathcal{C}) = \frac{1}{m} \sum_{(u,v) \in E} \delta(c(u), c(v)), \text{ where } \delta \text{ is the Kronecker delta function}$$

$$= \frac{1}{2m} \sum_{u,v \in V} a(u,v) \delta(c(u), c(v)), \text{ where } a(u,v) = 1 \text{ iff } (u,v) \text{ is an edge}$$

Initial Definition of Modularity



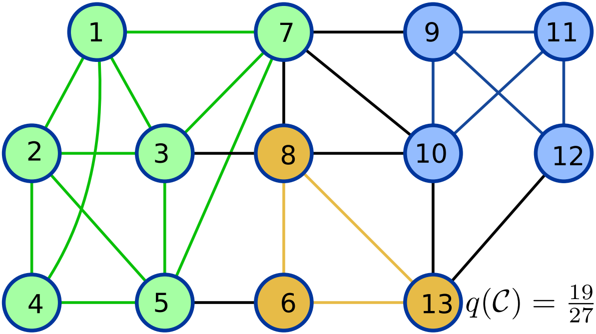
$q(\mathcal{C}) = \frac{19}{27}$

- How do we count the number of edges within modules?
- For every node $u \in V$, define $c(u)$ as the index of u's module.

$$q(\mathcal{C}) = \frac{1}{m} \sum_{(u,v) \in E} \delta(c(u), c(v)), \text{ where } \delta \text{ is the Kronecker delta function}$$

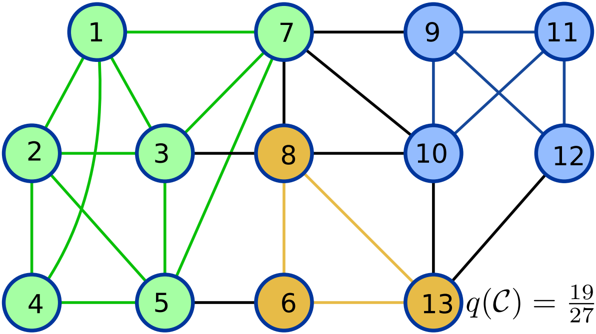
$$= \frac{1}{2m} \sum_{u,v \in V} a(u,v)\delta(c(u), c(v)), \text{ where } a(u,v) = 1 \text{ iff } (u,v) \text{ is an edge}$$

Optimising Modularity



$$q(\mathcal{C}) = \frac{1}{2m} \sum_{u,v \in V} a(u,v)\delta(c(u), c(v))$$

Optimising Modularity

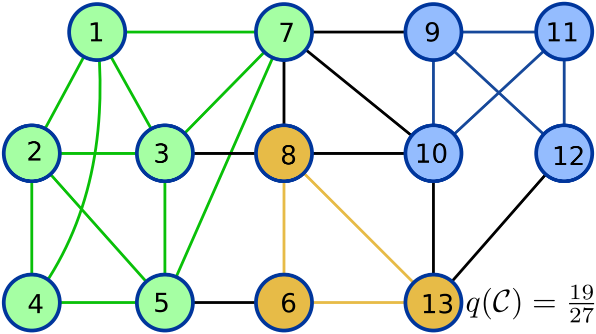


$q(\mathcal{C}) = \frac{19}{27}$

$$q(\mathcal{C}) = \frac{1}{2m} \sum_{u,v \in V} a(u,v)\delta(c(u), c(v))$$

- Should we maximise or minimise $q(\mathcal{C})$?

Optimising Modularity



$q(\mathcal{C}) = \frac{19}{27}$

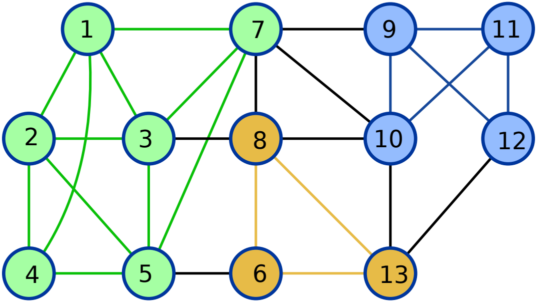
$$q(\mathcal{C}) = \frac{1}{2m} \sum_{u,v \in V} a(u,v)\delta(c(u), c(v))$$

- Should we maximise or minimise $q(\mathcal{C})$? Maximise it.
- If we place all nodes in G in a single cluster, $q(\mathcal{C}) = 1$!

Two Criteria for High Quality Partitions

1. Nodes are in highly cohesive modules, i.e., nodes within the same module will be strongly connected with each other.
2. The amount of intramodule connectivity in a good partition will be greater than expected by chance, as defined by a network in which edges are placed between nodes at random.
3. Proposed by Newman and Girvan, 2004.

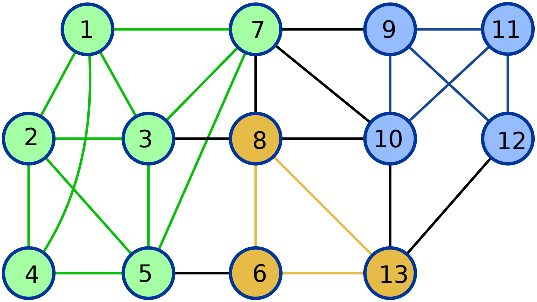
Final Definition of Modularity



$$q(\mathcal{C}) = \frac{1}{2m} \sum_{u,v \in V} \left(a(u,v) - \frac{d(u)d(v))}{2m} \right) \delta(C(u), C(v))$$

- What is the range of $q(\mathcal{C})$?

Final Definition of Modularity



$$q(\mathcal{C}) = \frac{1}{2m} \sum_{u,v \in V} \left(a(u,v) - \frac{d(u)d(v))}{2m} \right) \delta(C(u), C(v))$$

- What is the range of $q(\mathcal{C})$? Between -1/2 and 1.
  - $q(\mathcal{C}) > 0$: $\mathcal{C}$ has higher intramodule connectivity than expected by chance from configuration model.
  - $q(\mathcal{C}) = 0$: $\mathcal{C}$ has same intramodule connectivity as expected in a random graph.
  - $q(\mathcal{C}) < 0$: $\mathcal{C}$ has no modular structure.

Limitations of Modularity

- Modularity generally increases as number of nodes and modules in a graph increase.
- Many very similar partitions have similar values of q.
- Modularity has a resolution limit: small modules may be combined simply to increase q.
- Random graph model is quite simple: assumes every node has an equal probability of connecting to every other node.

Limitations of Modularity

- Modularity generally increases as number of nodes and modules in a graph increase.
- Many very similar partitions have similar values of q.
- Modularity has a resolution limit: small modules may be combined simply to increase q.
- Random graph model is quite simple: assumes every node has an equal probability of connecting to every other node.
- Many alternatives proposed to address these limitations.

Greedy Algorithm

- Proposed by Newman, 2004.

1. Start with every node in its own module.
2. While there are at least two modules
   1. Compute the pair of modules whose merger will result in the largest increase or smallest decrease in q.
   2. Merge this pair of modules into one.
3. Return the clustering with the largest value of q.

Greedy Algorithm

- Proposed by Newman, 2004.

1. Start with every node in its own module.
2. While there are at least two modules
    1. Compute the pair of modules whose merger will result in the largest increase or smallest decrease in q.
    2. Merge this pair of modules into one.
3. Return the clustering with the largest value of q.

- Hierarchical clustering algorithm built directly around maximisation of q.
- Allows q to decrease to preserve the principle of hierarchical clustering.
- Why is the algorithm "greedy"?

Greedy Algorithm

- Proposed by Newman, 2004.

1. Start with every node in its own module.
2. While there are at least two modules
   1. Compute the pair of modules whose merger will result in the largest increase or smallest decrease in q.
   2. Merge this pair of modules into one.
3. Return the clustering with the largest value of q.

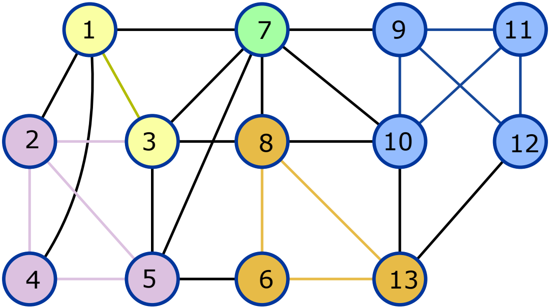
- Hierarchical clustering algorithm built directly around maximisation of q.
- Allows q to decrease to preserve the principle of hierarchical clustering.
- Why is the algorithm "greedy"? Merging of two modules cannot be undone.

Louvain Algorithm: Phase 1

- Proposed by Blondel *et al.*, 2008.

1. Start with every node in its own module.
2. For every node $u \in V$ and every neighbour v of u, evaluate the change in q when we remove u from its module and add it to v's module.
3. Move u to that neighbour's module for which increase in q is largest.
4. Repeat the previous two steps until q does not increase.

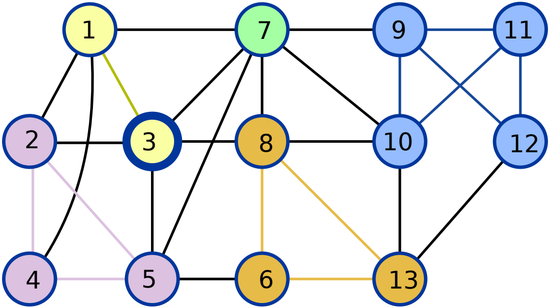
Louvain Algorithm: Phase 1



- Proposed by Blondel *et al.*, 2008.

1. Start with every node in its own module.
2. For every node $u \in V$ and every neighbour v of u, evaluate the change in q when we remove u from its module and add it to v's module.
3. Move u to that neighbour's module for which increase in q is largest.
4. Repeat the previous two steps until q does not increase.

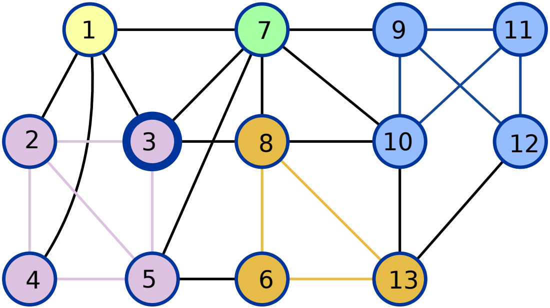
Louvain Algorithm: Phase 1



- Proposed by Blondel *et al.*, 2008.

1. Start with every node in its own module.
2. For every node $u \in V$ and every neighbour v of u, evaluate the change in q when we remove u from its module and add it to v's module.
3. Move u to that neighbour's module for which increase in q is largest.
4. Repeat the previous two steps until q does not increase.

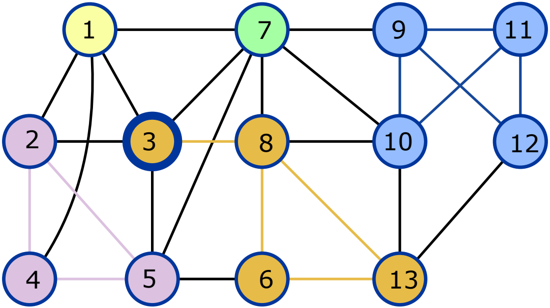
Louvain Algorithm: Phase 1



- Proposed by Blondel *et al.*, 2008.

1. Start with every node in its own module.
2. For every node $u \in V$ and every neighbour v of u, evaluate the change in q when we remove u from its module and add it to v's module.
3. Move u to that neighbour's module for which increase in q is largest.
4. Repeat the previous two steps until q does not increase.

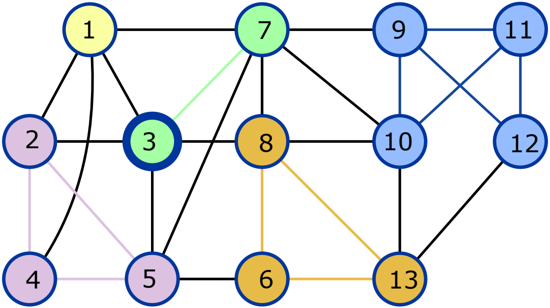
Louvain Algorithm: Phase 1



- Proposed by Blondel *et al.*, 2008.

1. Start with every node in its own module.
2. For every node $u \in V$ and every neighbour v of u, evaluate the change in q when we remove u from its module and add it to v's module.
3. Move u to that neighbour's module for which increase in q is largest.
4. Repeat the previous two steps until q does not increase.

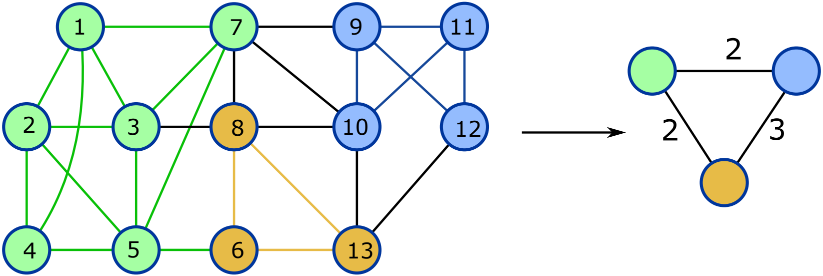
Louvain Algorithm: Phase 1



- Proposed by Blondel *et al.*, 2008.

1. Start with every node in its own module.
2. For every node $u \in V$ and every neighbour v of u, evaluate the change in q when we remove u from its module and add it to v's module.
3. Move u to that neighbour's module for which increase in q is largest.
4. Repeat the previous two steps until q does not increase.

Louvain Algorithm: Phase 2



1. Construct a new graph where every module is a node and a weighted edge represents (multiple) connections between two modules.
2. Repeat Phases 1 and 2 until no further gains in q are possible.

Louvain Algorithm: Pseudocode

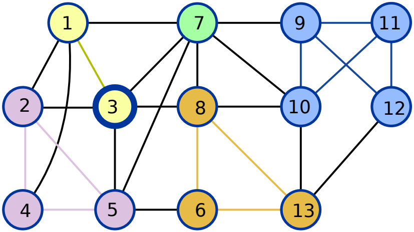
```
function LOUVAIN(Graph G, Partition 𝒫)
    do
        𝒫 ← MOVENODES(G, 𝒫)                    ▷ Move nodes between communities
        done ← |𝒫| = |V(G)|                    ▷ Terminate when each community consists of only one node
        if not done then
            G ← AGGREGATEGRAPH(G, 𝒫)           ▷ Create aggregate graph based on partition 𝒫
            𝒫 ← SINGLETONPARTITION(G)          ▷ Assign each node in aggregate graph to its own community
        end if
    while not done
    return flat*(𝒫)
end function
```

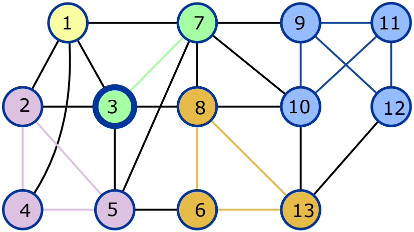
```
function AGGREGATEGRAPH(Graph G, Partition 𝒫)
    V ← 𝒫                                                  ▷ Communities become nodes in aggregate graph
    E ← {(C, D) | (u, v) ∈ E(G), u ∈ C ∈ 𝒫, v ∈ D ∈ 𝒫}    ▷ E is a multiset
    return GRAPH(V, E)
end function

function SINGLETONPARTITION(Graph G)
    return {{v} | v ∈ V(G)}                                ▷ Assign each node to its own community
end function
```

Louvain Algorithm: Pseudocode

function MoveNodes(Graph G, Partition $\mathcal{P}$)
    do
        $\mathcal{H}_{\text{old}} = \mathcal{H}(\mathcal{P})$
        for $v \in V(G)$ **do** ▷ Visit nodes (in random order)
            $C' \leftarrow \arg\max_{C \in \mathcal{P} \cup \emptyset} \Delta\mathcal{H}_{\mathcal{P}}(v \mapsto C)$ ▷ Determine best community for node v
            if $\Delta\mathcal{H}_{\mathcal{P}}(v \mapsto C') > 0$ **then** ▷ Perform only strictly positive node movements
                $v \mapsto C'$ ▷ Move node v to community C'
            end if
        end for
    while $\mathcal{H}(\mathcal{P}) > \mathcal{H}_{\text{old}}$ ▷ Continue until no more nodes can be moved
    return $\mathcal{P}$

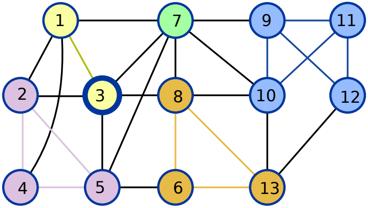
Louvain Algorithm: Efficiency

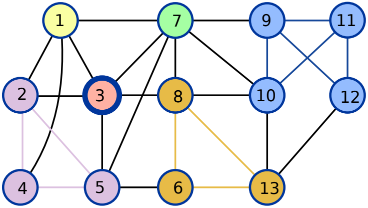




- Efficient calculation of change in q upon swapping makes this algorithm very fast.
- Calculate change in modularity when we move node i to neighbour j's community in two steps:
  1. Remove i from its community and move it to an isolated community.
  2. Merge this new community with j's community.

Louvain Algorithm: Moving Node *i* Out

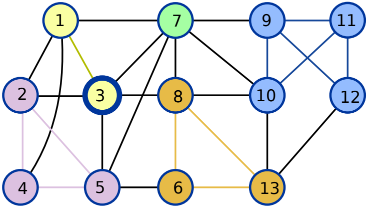


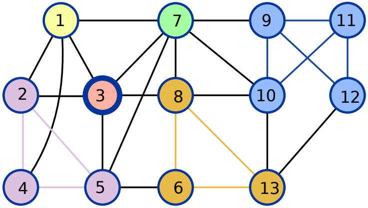


$$q(\mathcal{C}) = \frac{1}{2m} \sum_{u,v \in V} \left(a(u,v) - \frac{d(u)d(v))}{2m} \right) \delta(C(u), C(v))$$

- In the first step, for which node pairs does $\delta(C(u), C(v))$ change?

Louvain Algorithm: Moving Node i Out



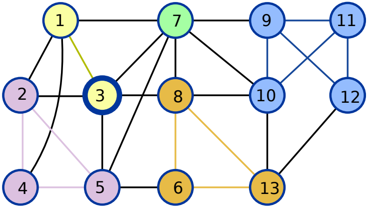


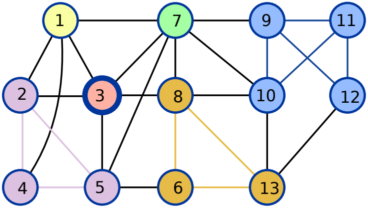
$$q(\mathcal{C}) = \frac{1}{2m} \sum_{u,v \in V} \left(a(u,v) - \frac{d(u)d(v))}{2m} \right) \delta(C(u), C(v))$$

- In the first step, for which node pairs does $\delta(C(u), C(v))$ change? Only if u or v equals i.

$$\Delta(q(\mathcal{C})) = -\frac{2}{2m} \sum_{u \in C(i)} \left(a(u,i) - \frac{d(u)d(i))}{2m} \right)$$
$$= -\frac{1}{m} d(i, C(i)) + \frac{d(i)}{2m^2} \sum_{u \in C(i)} d(u)$$

Louvain Algorithm: Moving Node *i* Out





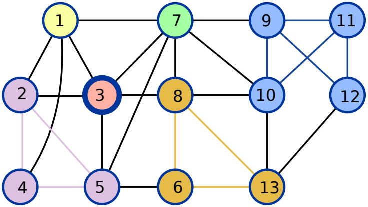
$$q(\mathcal{C}) = \frac{1}{2m} \sum_{u,v \in V} \left(a(u,v) - \frac{d(u)d(v))}{2m} \right) \delta(C(u), C(v))$$

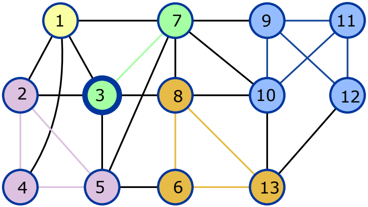
- In the first step, for which node pairs does $\delta(C(u), C(v))$ change? Only if u or v equals i.

$$\Delta(q(\mathcal{C})) = -\frac{2}{2m} \sum_{u \in C(i)} \left(a(u,i) - \frac{d(u)d(i))}{2m} \right)$$

$$= -\frac{1}{m} d(i, C(i)) + \frac{d(i)}{2m^2} \sum_{u \in C(i)} d(u)$$

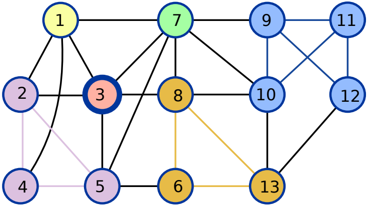
Louvain Algorithm: Moving Node i In

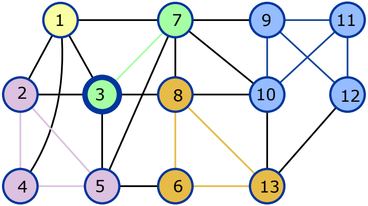




- In the second step, for which node pairs does $\delta(C(u), C(v))$ change?

Louvain Algorithm: Moving Node i In





- In the second step, for which node pairs does $\delta(C(u), C(v))$ change? Only if u or v equals i.
- Alternately, change in modularity is the negative of the change when we move i out of $C(j)$.

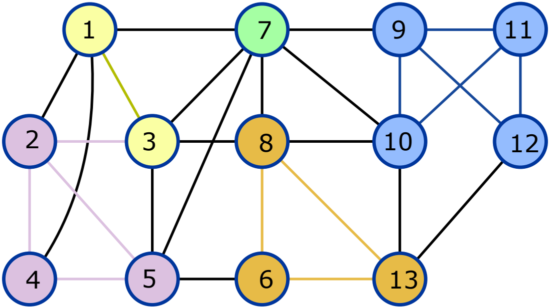
$$\Delta(q(\mathcal{C})) = \frac{1}{m} d(i, C(j)) - \frac{d(i)}{2m^2} \sum_{u \in C(j)} d(u)$$

- Compare to the formula in the Louvain paper and the Wikipedia page.

Comparisons to Other Algorithms

	Karate	Arxiv	Internet	Web nd.edu	Phone	Web uk-2005	Web WebBase 2001
Nodes/ links	34/77	9k/24k	70k/351k	325k/1M	2.04M/5.4M	39M/783M	118M/1B
CNM	0.38/0 s	0.772/3.6 s	0.692/799 s	0.927/5034 s	—/—	—/—	—/—
PL	0.42/0 s	0.757/3.3 s	0.729/575 s	0.895/6666 s	—/—	—/—	—/—
WT	0.42/0 s	0.761/0.7 s	0.667/62 s	0.898/248 s	0.553/367 s	—/—	—/—
Our algorithm	0.42/0 s	0.813/0 s	0.781/1 s	0.935/3 s	0.76/44 s	0.979/738 s	0.984/152 mn

Modularity Again

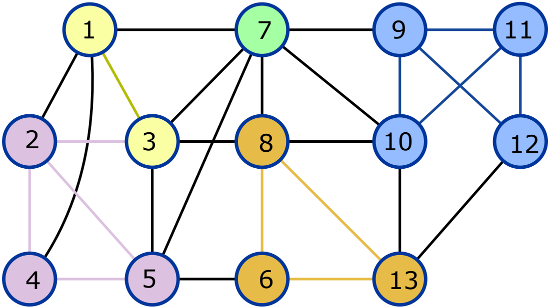


$$\mathcal{H} = \frac{1}{2m} \sum_c \left(e_c - \gamma \frac{K_c^2}{2m} \right)$$

$$q(\mathcal{C}) = \frac{1}{2m} \sum_{u,v \in V} \left(a(u,v) - \frac{d(u)d(v))}{2m} \right) \delta(C(u), C(v))$$

- In the formula for $\mathcal{H}$, what are the definitions of e_c and K_c?
- What role does γ play?

Constant Potts Model



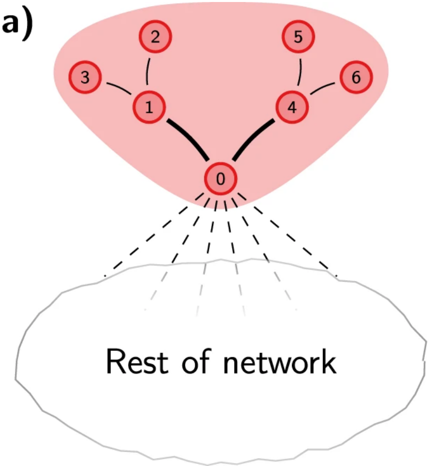
$$\mathcal{H} = \frac{1}{2m} \sum_c \left(e_c - \gamma \frac{K_c^2}{2m} \right)$$

$$\mathcal{H} = \sum_c \left(e_c - \gamma \binom{n_c}{2} \right)$$

- In the formula for $\mathcal{H}$, what is the definition of n_c?
- What role does γ play?

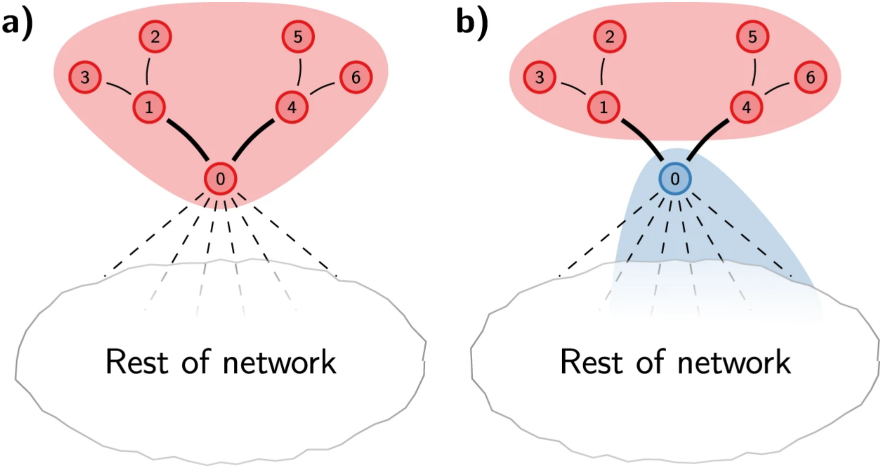
Problem with the Louvain Algorithm

Problem with the Louvain Algorithm



- The Louvain algorithm may find disconnected communities.

Problem with the Louvain Algorithm

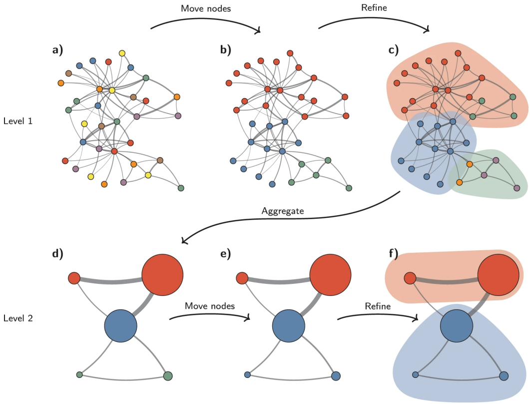


- The Louvain algorithm may find disconnected communities.
- In general, it may find arbitrarily badly connected communities.

Guarantees of the Louvain Algorithm

- At the end of each phase, communities are *well separated*, i.e., none can be merged to increase modularity.
- At the end of all phases, each node is optimally assigned.

Innovations in the Leiden Algorithm



- Includes a previously-introduced “smart local move”.
- Speeds up local moving of nodes.
- Moves nodes to “random” neighbours.
- Includes partition refinement before aggregation.

Leiden Algorithm: Pseudocode

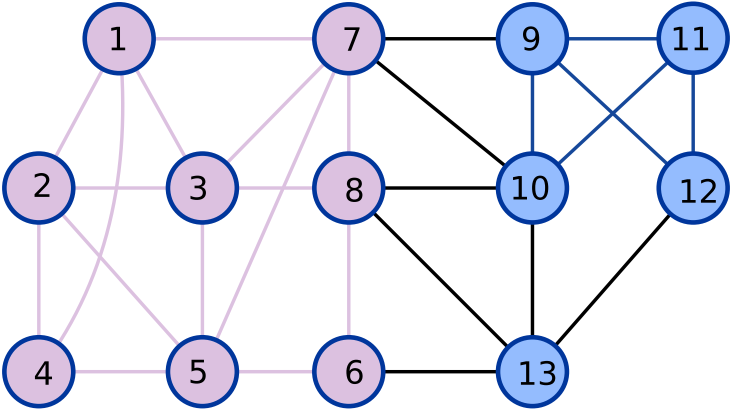
```
function LEIDEN(Graph G, Partition 𝒫)
    do
        𝒫 ← MOVENODESFAST(G, 𝒫)                              ▷ Move nodes between communities
        done ← |𝒫| = |V(G)|          ▷ Terminate when each community consists of only one node
        if not done then
            𝒫_refined ← REFINEPARTITION(G, 𝒫)                                ▷ Refine partition 𝒫
            G ← AGGREGATEGRAPH(G, 𝒫_refined)  ▷ Create aggregate graph based on refined partition 𝒫_refined
            𝒫 ← {{v | v ⊆ C, v ∈ V(G)} | C ∈ 𝒫}                         ▷ But maintain partition 𝒫
        end if
    while not done
    return flat*(𝒫)
end function

function AGGREGATEGRAPH(Graph G, Partition 𝒫)
    V ← 𝒫                                           ▷ Communities become nodes in aggregate graph
    E ← {(C, D) | (u, v) ∈ E(G), u ∈ C ∈ 𝒫, v ∈ D ∈ 𝒫}                          ▷ E is a multiset
    return GRAPH(V, E)
end function

function SINGLETONPARTITION(Graph G)
    return {{v} | v ∈ V(G)}                               ▷ Assign each node to its own community
end function
```

Louvain vs. Leiden: MoveNodes vs MoveNodesFast

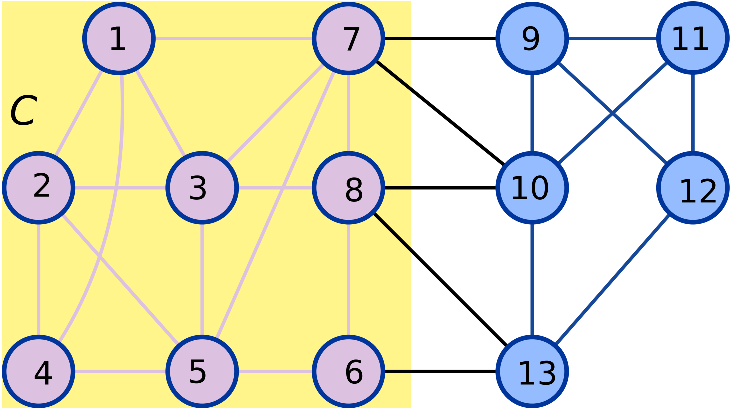
function MoveNodes(Graph G, Partition $\mathcal{P}$)
  do
    $\mathcal{H}_{\text{old}} = \mathcal{H}(\mathcal{P})$
    for $v \in V(G)$ **do** ▷ Visit nodes (in random order)
      $C' \leftarrow \arg\max_{C \in \mathcal{P} \cup \emptyset} \Delta\mathcal{H}_{\mathcal{P}}(v \mapsto C)$ ▷ Determine best community for node v
      if $\Delta\mathcal{H}_{\mathcal{P}}(v \mapsto C') > 0$ **then** ▷ Perform only strictly positive node movements
        $v \mapsto C'$ ▷ Move node v to community C'
      end if
    end for
  while $\mathcal{H}(\mathcal{P}) > \mathcal{H}_{\text{old}}$ ▷ Continue until no more nodes can be moved
  return $\mathcal{P}$

function MoveNodesFast(Graph G, Partition $\mathcal{P}$)
  $Q \leftarrow$ Queue($V(G)$) ▷ Make sure that all nodes will be visited (in random order)
  do
    $v \leftarrow Q$.remove() ▷ Determine next node to visit
    $C' \leftarrow \arg\max_{C \in \mathcal{P} \cup \emptyset} \Delta\mathcal{H}_{\mathcal{P}}(v \mapsto C)$ ▷ Determine best community for node v
    if $\Delta\mathcal{H}_{\mathcal{P}}(v \mapsto C') > 0$ **then** ▷ Perform only strictly positive node movements
      $v \mapsto C'$ ▷ Move node v to community C'
      $N \leftarrow \{u \mid (u, v) \in E(G), u \notin C'\}$ ▷ Identify neighbours of node v that are not in community C'
      Q.add($N - Q$) ▷ Make sure that these neighbours will be visited
    end if
  while $Q \neq \emptyset$ ▷ Continue until there are no more nodes to visit
  return $\mathcal{P}$
end function

Leiden: γ-Connected

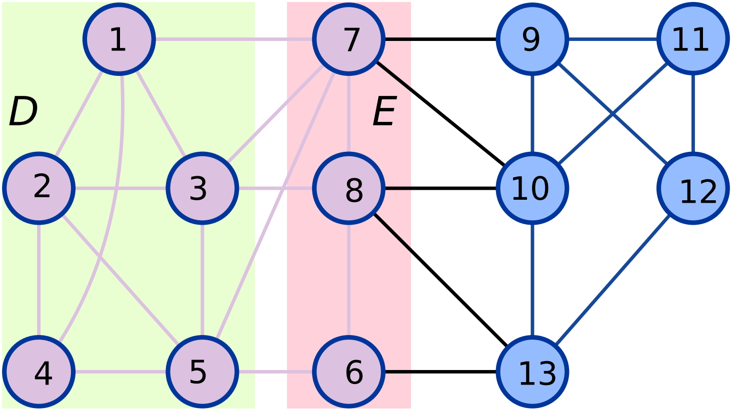


Leiden: γ-Connected



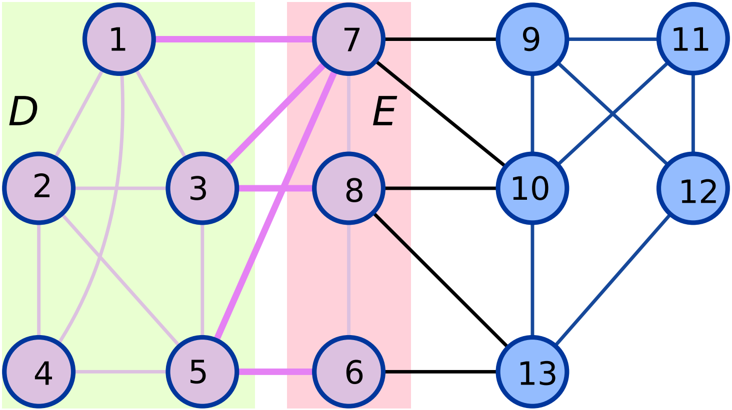
A module C is γ-connected ($\gamma = 0.3$)

Leiden: γ-Connected



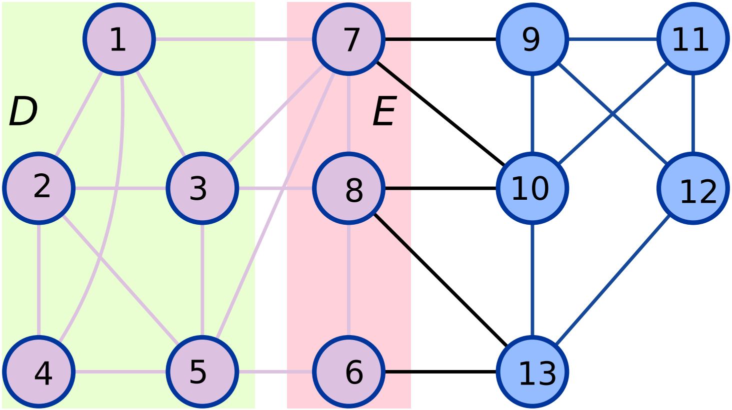
A module C is γ-connected ($\gamma = 0.3$) if it has two subsets D and E such

Leiden: γ-Connected



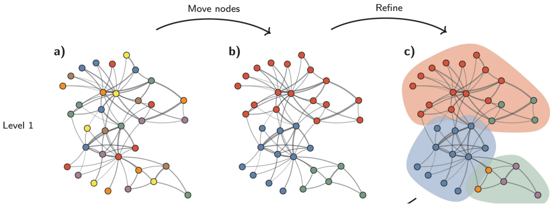
A module C is γ-connected ($\gamma = 0.3$) if it has two subsets D and E such $|E(C, D)| \geq \gamma |D||E|$

Leiden: γ-Connected



A module C is γ-connected ($\gamma = 0.3$) if it has two subsets D and E such $|E(C, D)| \geq \gamma |D||E|$ and D and E are also γ-connected.

Leiden: RefinePartition



function RefinePartition(Graph G, Partition $\mathcal{P}$)
  $\mathcal{P}_{\text{refined}} \leftarrow$ SingletonPartition(G) ▷ Assign each node to its own community
  for $C \in \mathcal{P}$ **do** ▷ Visit communities
    $\mathcal{P}_{\text{refined}} \leftarrow$ MergeNodesSubset($G, \mathcal{P}_{\text{refined}}, C$) ▷ Refine community C
  end for
  return $\mathcal{P}_{\text{refined}}$
end function

function MergeNodesSubset(Graph G, Partition $\mathcal{P}$, Subset S)
  $R = \{v \mid v \in S, E(v, S - v) \geq \gamma \|v\| \cdot (\|S\| - \|v\|)\}$ ▷ Consider only nodes that are well connected within subset S
  for $v \in R$ **do** ▷ Visit nodes (in random order)
    if v in singleton community **then** ▷ Consider only nodes that have not yet been merged
      $\mathcal{T} \leftarrow \{C \mid C \in \mathcal{P}, C \subseteq S, E(C, S - C) \geq \gamma \|C\| \cdot (\|S\| - \|C\|)\}$ ▷ Consider only well-connected communities
      $\Pr(C' = C) \sim \begin{cases} \exp\left(\frac{1}{\theta} \Delta \mathcal{H}_{\mathcal{P}}(v \mapsto C)\right) & \text{if } \Delta \mathcal{H}_{\mathcal{P}}(v \mapsto C) \geq 0 \\ 0 & \text{otherwise} \end{cases}$ for $C \in \mathcal{T}$ ▷ Choose random community C'
      $v \mapsto C'$ ▷ Move node v to community C'
    end if
  end for
  return $\mathcal{P}$
end function

Datasets

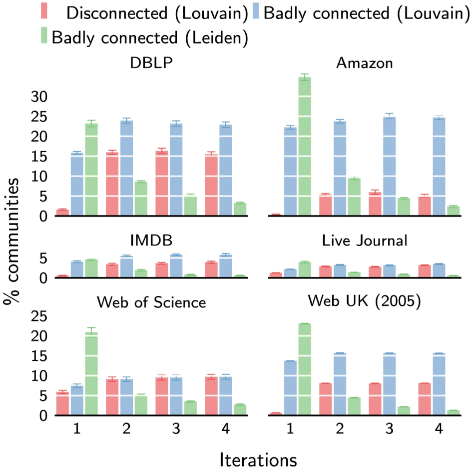
	Nodes	Degree	Max. modularity	
			Louvain	**Leiden**
DBLP	317,080	6.6	0.8262	0.8387
Amazon	334,863	5.6	0.9301	0.9341
IMDB	374,511	80.2	0.7062	0.7069
Live Journal	3,997,962	17.4	0.7653	0.7739
Web of Science	9,811,130	21.2	0.7911	0.7951
Web UK	39,252,879	39.8	0.9796	0.9801

Results: Badly Connected Communities

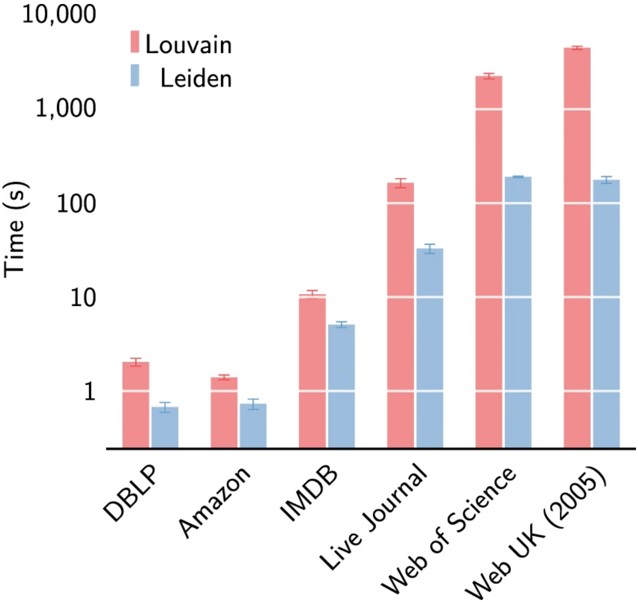
- A community C is *badly connected* if the Leiden algorithm run just on nodes in C can find smaller communities.

Results: Badly Connected Communities

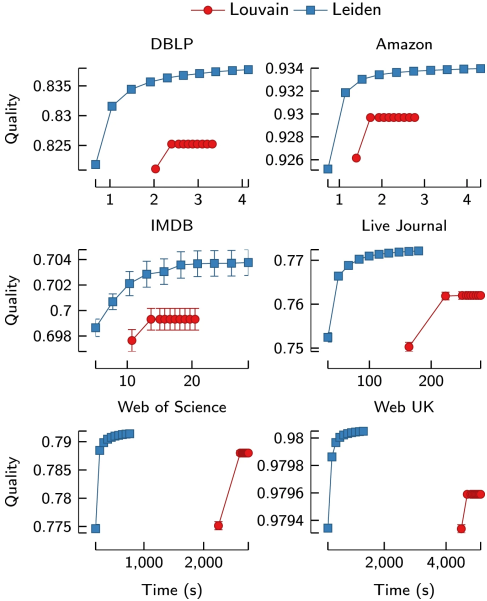
- A community C is *badly connected* if the Leiden algorithm run just on nodes in C can find smaller communities.



Results: Running Time



Results: Partition Quality



Summary

- The Louvain algorithm is very popular but may yield disconnected and badly connected communities.
- Iterating the algorithm worsens the problem.
- The Leiden algorithm guarantees γ-connected communities.
- It is also faster than the Louvain algorithm while computing communities with higher modularity.