VIETNAM NATIONAL UNIVERSITY - HO CHI MINH

UNIVERSITY OF SCIENCE

FINAL PROJECT PROPOSAL - TASK 1

Frequent Subgraph Mining, Graph Partition, Node Embedding, Graph Generation, Community Detection

Subject: Graph mining

Class: 21KHDL

Teacher: Lê Ngọc Thành

Lê Nhựt Nam

Students: Doãn Anh Khoa - 21127076

Đoàn Việt Hưng - 21127289

Đinh Bảo Trân - 21127454

Lê Nguyễn Phương Uyên - 21127476

Ho Chi Minh City - 2024

Table of Contents

1) Task Assignment

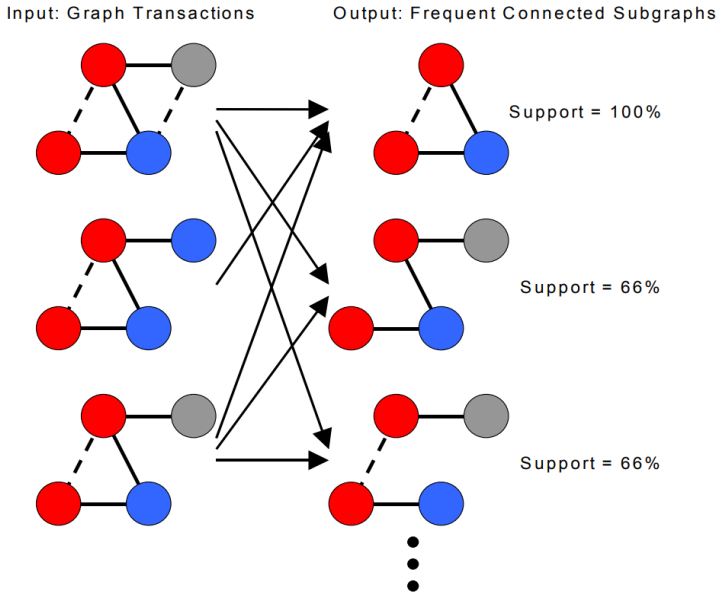
Task Name	Team Member	Status
Frequent Subgraph Mining - FSG	Trân	completed
Frequent Subgraph Mining - gSpan	Hưng	completed
Frequent Subgraph Mining - Subdue	Uyên	completed
Graph Partition - Kernighan–Lin	Khoa	completed
Graph Partition - Spectral Bisection	Hưng	completed
Node Embedding - Random walk	Trân	completed
Node Embedding - Node2vec	Uyên	completed
Graph Generation - Watts-Strogatz	Hưng	completed
Graph Generation - Barabási-Albert	Khoa	completed
Community Detection - Girvan-Newman	Trân	completed
Community Detection - Spectral Clustering	Uyên	completed
Task1 Report	Khoa	completed

2) Frequent Subgraph Mining

2.1 Introduction

- Frequent Subgraph Mining (FSM) is a data mining technique used to discover recurring patterns, or subgraphs, within a large set of graphs.
- This method is particularly useful in domains where data is naturally represented as graphs, such as social networks, biological networks, chemical compounds, and more.
- Key Concepts in Frequent Subgraph Mining:
  - *Graph Representation*: Data is represented as a collection of graphs, with nodes (vertices) and edges.
  - *Subgraph*: A smaller graph derived from a larger graph by selecting a subset of nodes and edges.
  - *Frequent Subgraph*: A subgraph that appears in a significant number of graphs within the dataset, meeting a specified "support" threshold.
  - *Support*: The proportion of graphs in the dataset that contain a particular subgraph.
  - *Mining Process*: Involves identifying all subgraphs that meet or exceed the support threshold by generating candidate subgraphs and checking their frequency.
- Parameters:

- **Input**:
  - *Graph Dataset*: A collection of graphs (e.g., molecular structures, social networks).
  - *Support Threshold*: Minimum frequency a subgraph must meet to be considered frequent.
- **Output**:
  - *Frequent Subgraphs*: ubgraphs that meet or exceed the support threshold.
  - *Support Values*: For each frequent subgraph, the support value shows the proportion or count of graphs in the dataset where the subgraph appears.



- Algorithms for FSM:
  - **Apriori-based Methods**: These are extensions of the Apriori algorithm used in association rule mining, adapted to work with graphs.
    - *FSG (Frequent Subgraph Mining)*: Extends the Apriori principle to graphs by generating candidate subgraphs through joining smaller frequent subgraphs.
    - *AGM (Apriori-based Graph Mining)*: Generates candidate subgraphs by expanding existing frequent subgraphs in a breadth-first manner.
  - **Pattern-Growth Methods**: These methods expand subgraphs by adding one edge at a time and checking the support of the resulting subgraph.
    - *gSpan (Graph-based Substructure Pattern Mining)*: Uses depth-first search and lexicographic ordering to efficiently mine frequent subgraphs without candidate generation.
    - *FFSM (Fast Frequent Subgraph Mining)*: Combines pattern-growth with canonical labeling to reduce the number of isomorphism checks.

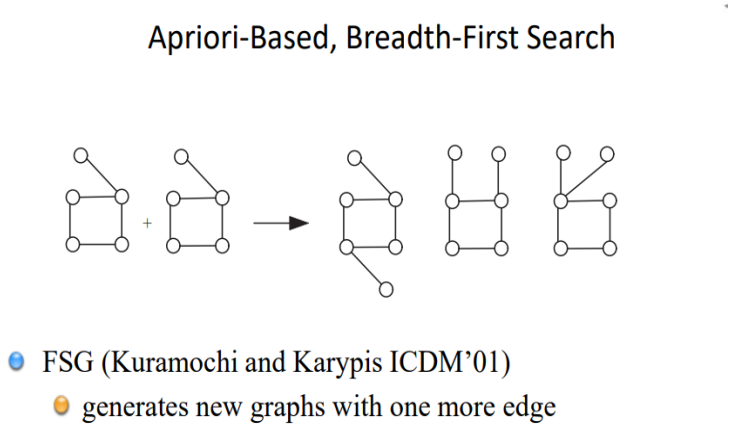
– **Other Notable Methods**:

* *SUBDUE*: Employs a heuristic approach to find subgraphs that best compress the graph's description while preserving its structure.
* *CHARM*: Uses constraints to limit the search space and find frequent subgraphs efficiently.

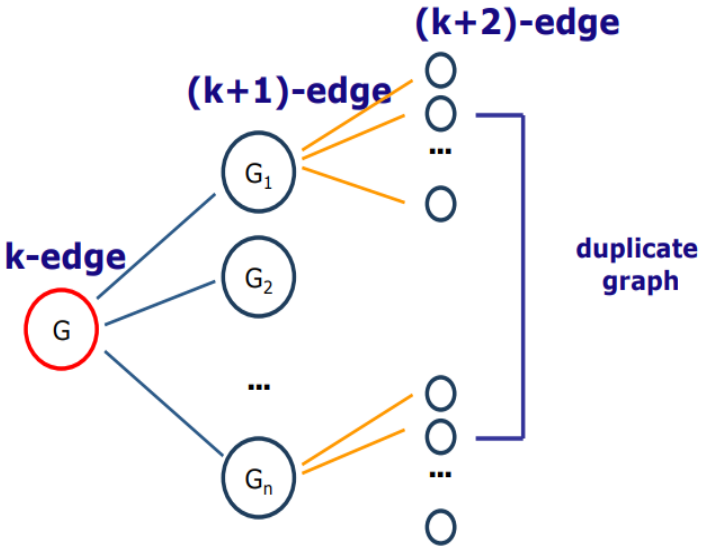
2.2 Common pattern finding method based on Apriori - FSG

2.2.1 Algorithm Description

- The FSG algorithm *[6]* is based on the Apriori principle, meaning that if a subgraph is not frequent, then all subgraphs extended from it (by adding an edge) will also not be frequent.
- Therefore, it is sufficient to check the frequent subgraphs of smaller size before extending them.

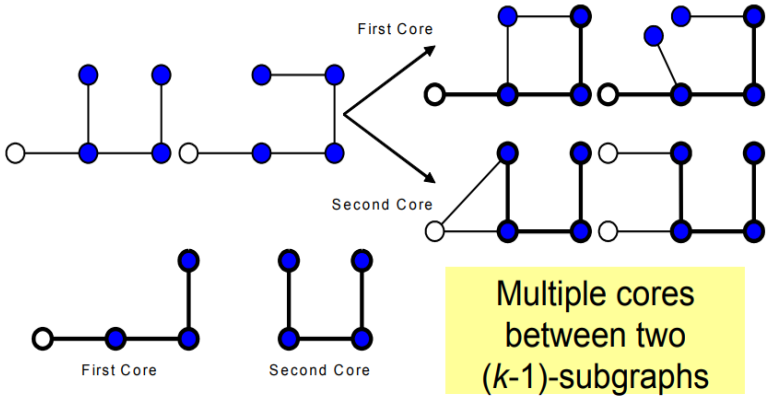


- Starting with a subgraph with k edges(k-edge), the algorithm generates new subgraphs by adding an edge to the existing subgraph.
- In each step, the algorithm extends the k-edge subgraphs into $G_1, G_2, \ldots, G_n$ with $k+1$ edges ($k+1$-edge)
- This extension process is performed sequentially, meaning that a k-edge subgraph generates subgraphs with k+1 edges, which then continue to generate subgraphs with k+2 edges, and so on.

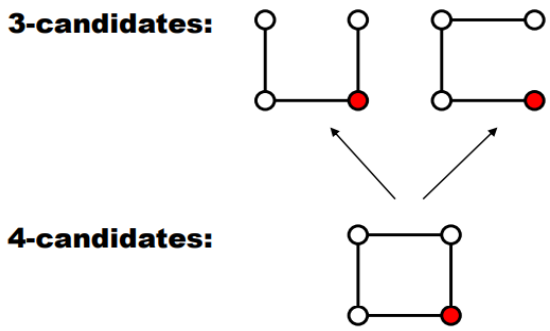


2.2.2 Steps of the FSG Algorithm

- **Step 1: Initialization - Find F1 and F2**
  - F_1: This is the set of all 1-subgraphs (subgraphs containing one edge) with a frequency greater than or equal to the minimum support threshold (#minSup).
  - F_2: This is the set of all 2-subgraphs (subgraphs containing two edges) with a frequency greater than or equal to #minSup.
- **Step 2: Candidate Generation**
  - In this step, the set C_k (candidate k-edge subgraphs) is created from F_{k-1}.
  - Each k-edge subgraph in C_k is generated by combining two $(k-1)$-edge subgraphs from F_{k-1} that share the same $(k-2)$ edges and adding exactly one new edge.
  - Specifically, two $(k-1)$-edge subgraphs g_1 and g_2 from F_{k-1} can be combined to form a k-edge subgraph if:
    - g_1 and g_2 share $(k-2)$ common edges (they have the same structure when one edge is removed).
    - Adding a new edge (the difference between g_1 and g_2) to g_1 or g_2 will create a k-edge subgraph.



- **Step 3: Candidates Pruning**
  - A necessary condition for a candidate k-edge subgraph to be frequent is that all its $(k-1)$-edge subgraphs must be frequent.
  - This step removes k-edge subgraphs in C_k that have at least one $(k-1)$-edge subgraph not present in F_{k-1}.



- **Step 4: Frequency Counting**
  - Scan through the data to count the occurrences of the subgraphs in C_k. These subgraphs are then compared with #minSup.
- **Step 5: Repeat Steps 2, 3, and 4 iteratively**
  - For $k = 3$; $F_{k-1} \neq \emptyset$; $k++$: This loop starts from $k = 3$ and continues until F_{k-1} (the set of $(k-1)$-edge subgraphs) becomes empty (i.e., no more frequent subgraphs are found).
- **Step 6: Return Results**

– Return $F_1 \cup F_2 \cup \ldots \cup F_k$ $(= F)$: After the loop terminates, the algorithm returns a set F containing all frequent subgraphs in the original dataset.

2.2.3 Pseudo Code

Algorithm 1 Frequent Subgraph Mining (FSG)

function FSG($(\mathcal{D}, \text{minSup})$)
  Initialize F_1 as the set of all frequent 1-edge subgraphs:
  $F_1 \leftarrow \{\text{all edges in } \mathcal{D} \text{ that appear in at least minSup graphs}\}$
  $F \leftarrow F_1$
  for $k = 2$ **to** ∞ **do**
    $C_k \leftarrow \text{CandidateGeneration}(F_{k-1})$
    $C_k \leftarrow \text{Prune}(C_k, F_{k-1})$
    Count occurrences of each candidate subgraph in C_k
    Find F_k:
    $F_k \leftarrow \{c \in C_k \mid \text{frequency}(c) \geq \text{minSup}\}$
    if $F_k = \emptyset$ **then**
      break
    end if
    $F \leftarrow F \cup F_k$
  end for
  return F
end function

Algorithm 2 Candidate Generating (FSG)

function CandidateGeneration(F_{k-1})
  Initialize $C_k \leftarrow \{\}$
  for each pair of subgraphs $(g_1, g_2) \in F_{k-1}$ **do**
    if g_1 and g_2 share a common $(k-2)$-subgraph **then**
      Create a new subgraph by combining g_1 and g_2 with one additional edge
      Add the new subgraph to C_k
    end if
  end for
  return C_k
end function

Algorithm 3 Candidate Prunning (FSG)

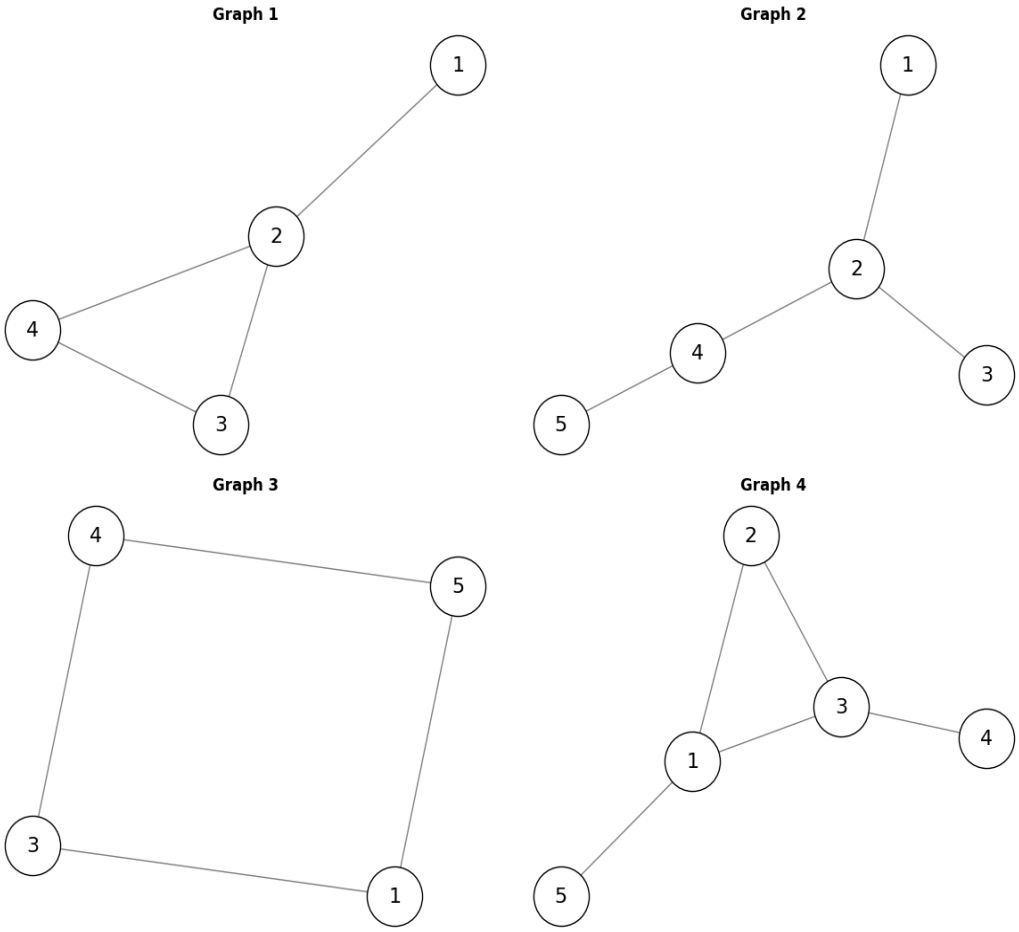
function Prune(C_k, F_{k-1})
  Initialize $C'_k \leftarrow \{\}$
  for each candidate $c \in C_k$ **do**
    if all $(k-1)$-subgraphs of c are in F_{k-1} **then**
      Add c to C'_k
    end if
  end for
  return C'_k
end function

2.2.4 Example of the FSG Algorithm

Consider the following graph dataset:

- **Graph 1**: G_1 with edges $\{(1,2),(2,3),(2,4),(3,4)\}$
- **Graph 2**: G_2 with edges $\{(1,2),(2,4),(4,5),(2,3)\}$
- **Graph 3**: G_3 with edges $\{(1,5),(3,4),(4,5),(1,3)\}$

- **Graph 4**: G_4 with edges $\{(2,3),(3,1),(1,5),(3,4),(1,2)\}$
- And consider the support value with **#minSup** $= 2$.



Step 1: Initialization

- *Find F_1: Identify all 1-edge subgraphs (edges) that appear in at least 2 graphs.*
  - Count of edges:

Edges	Number of appearances
$(1,2)$	3
$(1,3)$	2
$(1,4)$	0
$(1,5)$	2
$(2,3)$	3
$(2,4)$	2
$(2,5)$	0
$(3,4)$	3
$(3,5)$	0
$(4,5)$	2

– Keep only the edges with a frequency greater than the #minSup = 2

$$\Rightarrow F_1 = \{(1,2),(1,3),(1,5),(2,3),(2,4)(3,4),(4,5)\}$$

- *Find F_2: Identify all 2-edge subgraphs (paths of length 2) that appear in at least 2 graphs.*

  – Table of edge pairs satisfying the condition of having appearances $> \#minSup = 2$:

Edges	Number of appearances
$(1,2)-(2,4)$	2
$(3,2)-(2,4)$	2
$(1,2)-(2,3)$	3
$(2,3)-(3,4)$	2
$(3,1)-(1,5)$	2

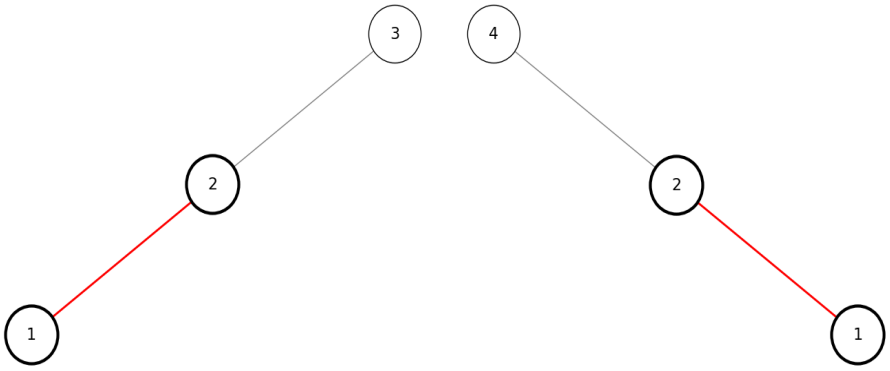
$$\Rightarrow F_2 = \{(1,2)-(2,4),(3,2)-(2,4),(1,2)-(2,3),(2,3)-(3,4),(3,1)-(1,5)\}$$

Step 2: Candidate Generation

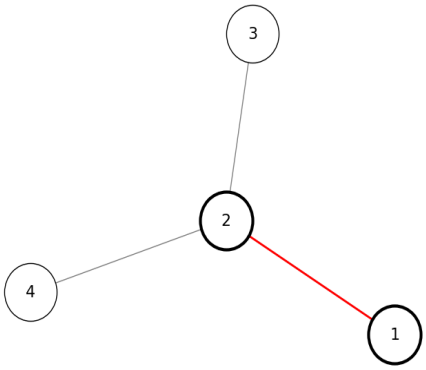
Generate new candidates array (C_k) based on previous graphs in F_{k-1} from step 1.

- Figure the two graphs $(1,2)-(2,4)$ and $(1,2)-(2,3)$:

  – We can see (1,2) is a core-edge (the same edge between 2 graph)



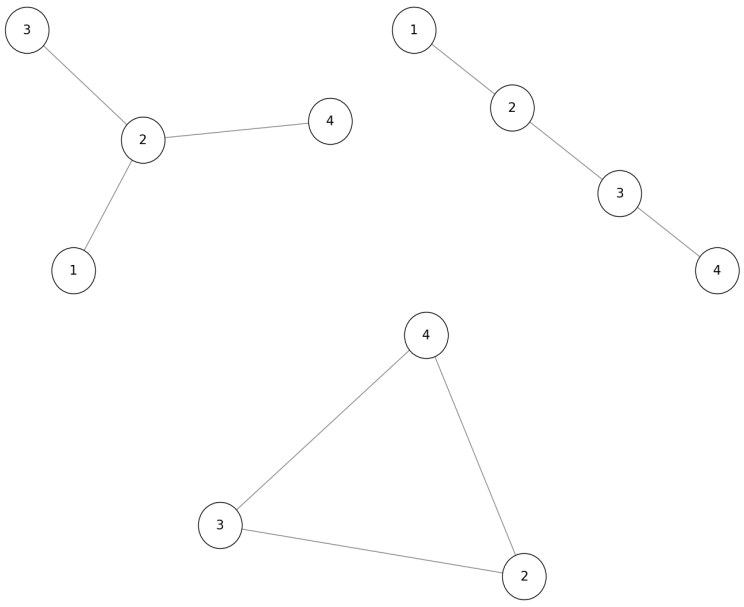
– We can generate a candidate based on the core-edge (common edge) and add the different edges from the two graphs.



⇒ Complete creating new candidate $G = (1,2) - (2,4) - (2,3) \in C_3$

- Do the same with other pairs, we'll get:

$$C_3 = \{(1,2) - (2,4) - (2,3), (1,2) - (2,3) - (3,4), (2,4) - (2,3) - (3,4)\}$$



Step 3: Candidates Pruning

Remove candidates from C_k that do not have all their $(k-1)$-edge subgraphs in F_{k-1}.

- Check subgraphs in C_3:
  - $\{(1,2)-(2,4)-(2,3)\}$ has all its 2-edge subgraphs in F_2.
  - $\{(1,2)-(2,3)-(3,4)\}$ also has all its 2-edge subgraphs in F_2.
  - $\{(2,4)-(2,3)-(3,4)\}$ also has all its 2-edge subgraphs in F_2.

  $\Rightarrow C_3$ remains unchanged.

Step 4: Frequency Counting

Count the number of graphs each candidate appears in.

Edges	Number of appearances
$(1,2)-(2,4)-(2,3)$	2
$(1,2)-(2,3)-(3,4)$	2
$(2,4)-(2,3)-(3,4)$	1

- It can be seen that the graph $(2,4)-(2,3)-(3,4)$ does not satisfy the condition of having an appearance frequency greater than #minSup.

  $\Rightarrow$ Remove graph $\{(2,4)-(2,3)-(3,4)\}$ from C_3 and save to F_3

  $\Rightarrow F_3 = \{(1,2)-(2,4)-(2,3), (1,2)-(2,3)-(3,4)\}$

Step 5: Repeat

Check if candidates appear in at least #minSup graphs, so they are frequent subgraphs.

- Repeat steps 2, 3, and 4 iteratively in the following loop:

$$\text{For } k=3;\ F_{k-1} \neq \emptyset;\ k++$$

- This loop starts from $k=3$ and continues until F_{k-1} (the set of $(k-1)$-edge subgraphs) becomes empty (i.e., no more frequent subgraphs are found).

Step 6: Return Results

Return Frequent Subgraphs

- $F_1 \cup F_2 \cup F_3 = \{(1,2),(1,3),(1,5),(2,3),(2,4)(3,4),(4,5)\} \cup \{\{(1,2)-(2,4),(3,2)-(2,4),(1,2)-(2,3),(2,3)-(3,4),(3,1)-(1,5)\},\{(1,2)-(2,4)-(2,3),(1,2)-(2,3)-(3,4)\}\}$

  $\Rightarrow F = \{(1,2),(1,3),(1,5),(2,3),(2,4)(3,4),(4,5),(1,2)-(2,4),(3,2)-(2,4),(1,2)-(2,3),(2,3)-(3,4),(3,1)-(1,5),(1,2)-(2,4)-(2,3),(1,2)-(2,3)-(3,4)\}$

2.2.5 Advantages and Disadvantages of the FSG Algorithm

Advantages

1. *Based on the Apriori principle:*
   The FSG algorithm inherits the Apriori principle, meaning that if a subgraph is not frequent, then any extended subgraphs from it will also not be frequent. This significantly reduces the number of subgraphs that need to be considered, thereby improving computational efficiency.

2. *Scalability:*
   The FSG algorithm scales well to large datasets because it builds frequent subgraphs from smaller subgraphs step by step, reducing the creation of unnecessary subgraphs.

3. *Detection of complex graph structures:*
   FSG has the ability to detect complex structural patterns in graphs, which is useful in applications such as social network analysis, biological analysis, and other fields involving graph data.

4. *Efficiency with sparse graphs:*
   FSG is efficient with sparse graphs, where smaller subgraphs tend to be more frequent than larger subgraphs.

Disadvantages

1. *High computational cost:*
   Although FSG is more optimized than some other algorithms, the computation and storage of subgraphs during candidate generation and pruning can still be expensive, especially with large or dense graphs.

2. *Memory management challenges:*
   As the size and number of frequent subgraphs increase, memory requirements also increase, which can pose challenges in handling large datasets if effective memory management strategies are not employed.

3. *Dependency on the support threshold (minSup):*
   The results of the FSG algorithm are heavily dependent on the chosen support threshold (minSup). If the threshold is too high, many frequent subgraphs may be missed. Conversely, if the threshold is too low, the number of frequent subgraphs may become too large to manage.

4. *Not suitable for complex labeled graphs:*
   FSG works well with simple or unlabeled graphs. However, with complex labeled or weighted graphs, the algorithm may require adjustments or extensions, increasing computational complexity.

5. *Decreased performance with dense graphs:*

   For dense graphs, the number of subgraphs can grow very quickly, leading to a combinatorial explosion, which reduces the performance of the algorithm.

2.3 Depth-based common pattern finding method - gSpan

2.3.1 Algorithm Description

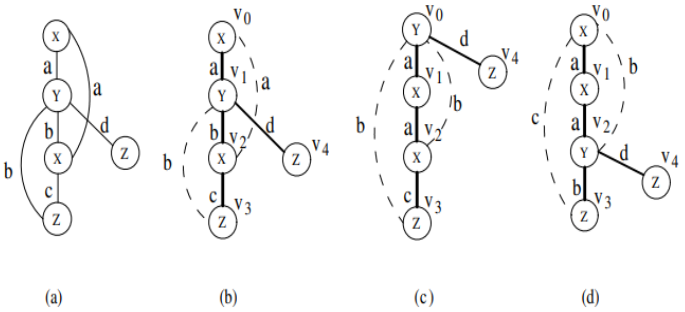
- The depth-based common pattern-finding method refers to techniques in data mining and graph analysis that involve exploring and identifying common patterns or substructures in data through a depth-first search strategy. This approach is often used in the context of frequent pattern mining, especially in graph databases.
- gSpan (Graph-based Substructure Pattern Mining) *[9]* is a powerful algorithm for mining meaningful and frequently occurring subgraph patterns in graph databases. This algorithm uses a pattern-growth method and a special data structure called DFS code to construct and extend graph patterns.
- The approach is based on:
  - Transforming the graph (2-dimensional) into a sequentially encoded edge representation using depth-first search (DFS) traversal.
  - Selecting the smallest DFS code.
- Challenges:
  - Combinatorial Explosion: The number of possible subgraphs in a graph database can grow exponentially with the size of the graphs and the number of graphs in the database, making the task computationally intensive.
  - Graph Isomorphism: Identifying identical subgraphs (isomorphic subgraphs) across different graphs is computationally complex, adding another layer of difficulty to the problem.

2.3.2 Step-by-step

gSpan addresses these challenges through a novel approach that avoids the generation of candidate subgraphs and effectively manages the complexity of graph isomorphism. The algorithm works in the following steps:

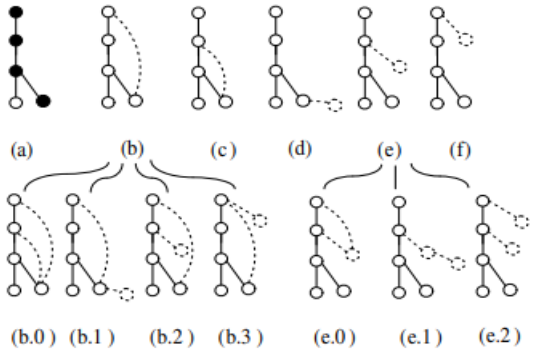
- DFS Code Generation
  - gSpan starts by systematically exploring subgraphs using a Depth-First Search (DFS) traversal of the graph. During this traversal, it generates a DFS code, which is a unique sequence representing the traversal order of the vertices and edges in the graph.

– This DFS code acts as a canonical label for each subgraph, allowing the algorithm to identify and differentiate subgraphs.



edge	(Fig 1b) α	(Fig 1c) β	(Fig 1d) γ
0	$(0,1,X,a,Y)$	$(0,1,Y,a,X)$	$(0,1,X,a,X)$
1	$(1,2,Y,b,X)$	$(1,2,X,a,X)$	$(1,2,X,a,Y)$
2	$(2,0,X,a,X)$	$(2,0,X,b,Y)$	$(2,0,Y,b,X)$
3	$(2,3,X,c,Z)$	$(2,3,X,c,Z)$	$(2,3,Y,b,Z)$
4	$(3,1,Z,b,Y)$	$(3,0,Z,b,Y)$	$(3,0,Z,c,X)$
5	$(1,4,Y,d,Z)$	$(0,4,Y,d,Z)$	$(2,4,Y,d,Z)$

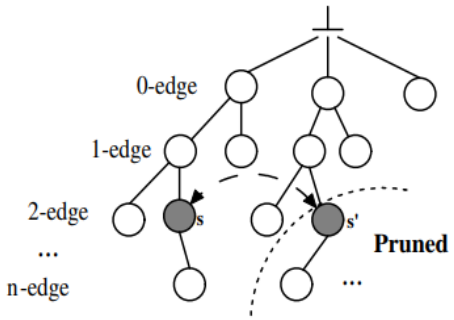
- Lexicographic Ordering
  - The algorithm introduces a lexicographic ordering of DFS codes. By establishing an order, gSpan ensures that it systematically explores subgraphs and avoids redundant exploration.
  - The smallest (or minimal) DFS code is used as the canonical form of a subgraph. If a subgraph's DFS code is not minimal, it is pruned from the search space, as it represents a redundant exploration path



- Frequent Subgraph Mining
  - gSpan extends the DFS code by adding edges incrementally, exploring larger and larger subgraphs.

- During each extension, the algorithm checks if the newly formed subgraph is frequent by counting its occurrences in the graph database.
- If the subgraph is frequent, the algorithm continues to extend it; if it is not, the subgraph is pruned from further exploration.

- Pruning: gSpan uses several pruning techniques to eliminate subgraphs that cannot be frequent, thereby reducing the search space and improving efficiency. For instance, if a subgraph is found to have a non-minimal DFS code, all its extensions are also non-minimal and can be pruned.



- **Pseudo code**

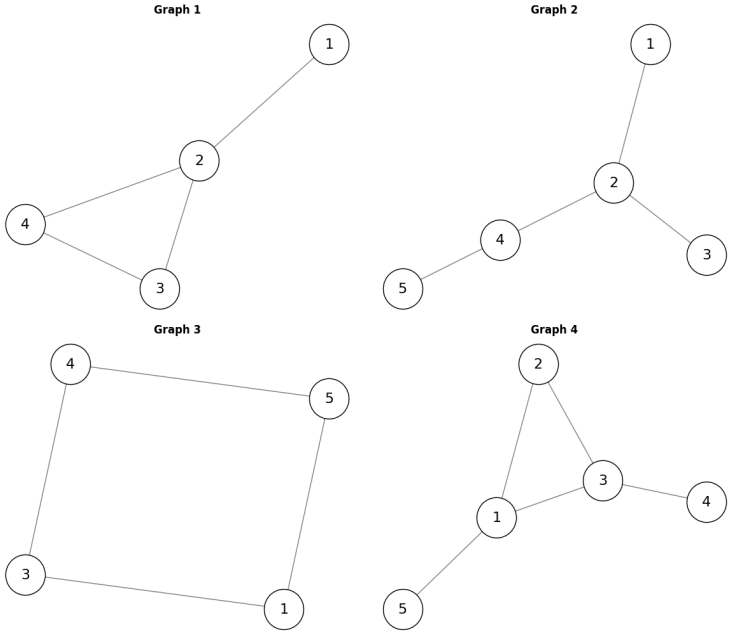
Algorithm 4 GraphSet Projection $(\mathcal{D}, S)$

```
1: function GraphSet Projection((D, S))
2:     Sort the labels in D by their frequency;
3:     Remove infrequent vertices and edges;
4:     Relabel the remaining vertices and edges;
5:     S^1 ← all frequent 1-edge graphs in D;
6:     sort S^1 in DFS lexicographic order;
7:     S ← S^1
8:     for each edge e ∈ D do
9:         initialize s with e, set s.D by graphs that contains e;
10:        Subgraph Mining(D, S, s);
11:        D ← D − e
12:        if |D| < minSup then
13:            break
14:        end if
15:    end for
16: end function
```

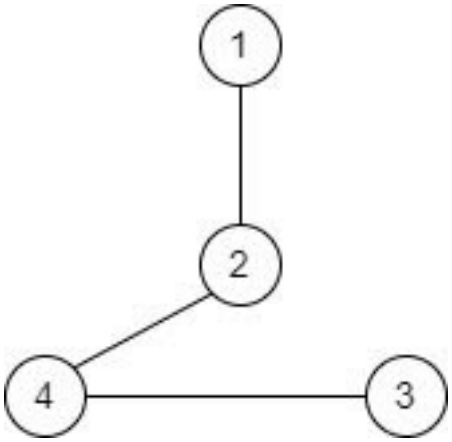
Algorithm 5 Subgraph_Mining($\mathcal{D}$, S,s)

```
1: if s ≠ min(s) then
2:     return ;
3: end if
4: S ← S ∪ {s};
5: Enumerate s in each graph in D and count its children;
6: for each c, c is s' child do
7:     if support(c) ≥ minSup then
8:         s ← c;
9:         Subgraph_Mining(D_s, S,s)
10:    end if
11: end for
```

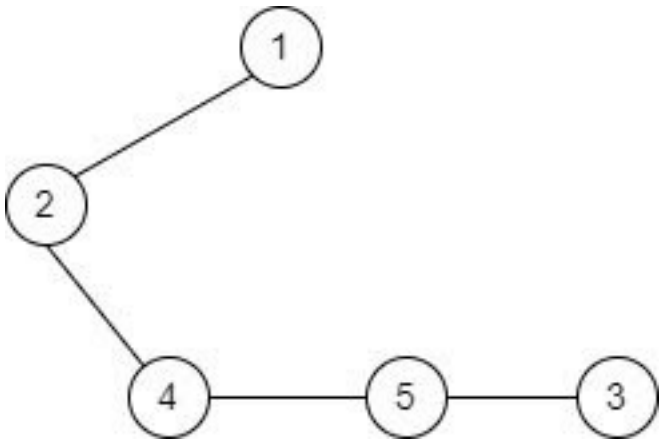
2.3.3 Example of the algorithm gSpan



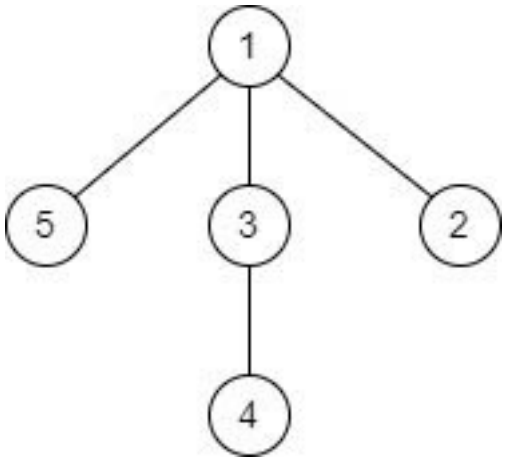
- **Step 1:** Extract Graph Information
  - Graph 1: Vertices: 1, 2, 3, 4, Edges: (1-2), (2-3), (3-4), (2-4)
  - Graph 2: Vertices: 1, 2, 3, 4, 5, Edges: (1-2), (2-3), (2-4), (4-5)
  - Graph 3: Vertices: 1, 3, 4, 5, Edges: (1-3), (3-4), (4-5), (5-1)
  - Graph 4: Vertices: 1, 2, 3, 4, 5, Edges: (1-2), (1-5), (1-3), (2-3), (3-4)
- **Step 2:** Apply gSpan - DFS Code Generation
  - DFS Traversal: For each graph, we perform a depth-first search to generate DFS codes. These codes will be sequences representing the traversal and connections.
  - DFS Code Representation:
    - Graph 1: DFS traversal might yield a code like 1-2-4-3.



* Graph 2: DFS traversal might yield 1-2-4-5-3.



* Graph 3: A possible DFS code is 1-5-4-3-1.
* Graph 4: DFS traversal might result in 1-5-3-4-2.



* Minimal DFS Code: We then compare DFS codes lexicographically to identify the minimal DFS code for each subgraph.

- **Step 3:** Identify Frequent Subgraphs
  - Identify Subgraphs: Starting from smaller subgraphs, identify those that are common across multiple graphs. For instance, the subgraph (1-2) might be a common pattern.
  - Check Frequency: Determine if a subgraph is frequent by counting its occurrence across all graphs. For example:
    * (1-2) appears in Graphs 1, 2, and 4.
    * (2-3) appears in Graphs 1, 2, and 4.
    * The subgraph (1-2-3-4) is present in Graphs 1 and 4.
- **Step 4:** Prune and Extend
  - Pruning:

    - If a subgraph is not frequent (i.e., does not meet the minimum support threshold), it and its extensions are pruned.
    - For instance, if we set the minimum support threshold equal to 3, subgraphs '(1-2-3-4)' found in only two graphs would be pruned.
  - Extension
    - Extend frequent subgraphs by adding edges to see if larger frequent subgraphs can be formed.
    - For example, starting from (1-2) in Graphs 1 and 4, we could extend to include (1-2-3) and then (1-2-3-4).

- **Step 5:** Output Frequent Subgraphs: After the algorithm completes, the frequent subgraphs across the input graphs would be listed, potentially including patterns like:
  - (1-2) as it appears frequently in multiple graphs.
  - (2-3) if it meets the frequency threshold.

2.3.4 Advantages and Disadvantages

- Advantages:
  - **Efficiency:** gSpan uses a depth-first search strategy, allowing it to explore graph patterns' search space efficiently. This results in faster identification of frequent subgraphs compared to some traditional methods.
  - **Pattern Growth:** The algorithm operates based on the idea of pattern growth, meaning it starts with smaller subgraphs and grows them step by step. This approach helps reduce the computational burden since it avoids generating a large number of candidate subgraphs at once.
  - **No Need for Candidate Generation:** gSpan does not require the explicit generation of candidate subgraphs. Instead, it generates frequent patterns directly, which can lead to significant performance improvements.
  - **Handling of Large Datasets:** gSpan is designed to handle large datasets effectively, making it suitable for real-world applications where graph data can be complex and voluminous.
  - **Support for Different Graph Types:** gSpan can be applied to various types of graphs (e.g., directed, undirected, labeled), providing flexibility for different domains and applications.
  - **Supports Isomorphism Checking:** The algorithm includes an efficient mechanism to check for isomorphism among subgraphs, which helps identify truly distinct patterns and reduces redundancy in the results.

- **Scalability:** The algorithm can scale well with increasing data sizes and complexity, making it an attractive option for large-scale graph mining tasks.
- **Widely Used:** gSpan has been extensively studied and applied in various fields, such as bioinformatics, chemical compound analysis, social network analysis, and more, demonstrating its practical utility and versatility.

- Disadvantages:
  - **Difficulty in handling large graphs:** gSpan may struggle when working with large or complex graphs, as the number of substructures can grow exponentially.
  - **Computationally intensive:** The algorithm can take a considerable amount of time to run, especially if you need to search for many substructures in large graphs.
  - **Sensitivity to parameters:** gSpan requires several parameters to be tuned, and selecting these parameters is not straightforward, which can affect the algorithm's performance.
  - **Inaccuracy of results:** In some cases, gSpan may not find all the substructures you desire, leading to results that are not entirely accurate.
  - **Challenges in scalability:** Extending gSpan to work with more complex data types or in different contexts can be challenging.
  - **Incompatibility with weighted graphs:** For weighted graphs, gSpan may not perform as effectively as it does with unweighted graphs.

2.4 Greedy frequent pattern finding method - Subdue

2.4.1 Algorithm Description

- Subdue *[5]* is a graph-based algorithm designed for the discovery of patterns in data, particularly in the form of graphs.

- Inputs to the Subdue system can be a single graph or a set of graphs. The graphs can be labeled or unlabeled. The DL of the input dataset G using substructure S can be calculated using the following formula,

$$I(S) + I(G|S)$$

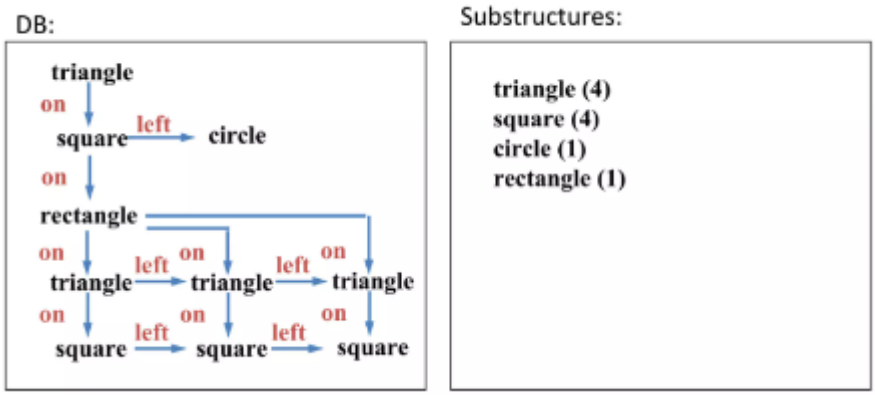
  where S is the substructure used to compress the dataset G. I(S) and I(G|S) represent the number of bits required to encode S and dataset G after S compresses G.

- Subdue outputs substructures that best compress the input dataset according to the Minimum Description Length (MDL) principle. The final output of Subdue is a set of substructures (subgraphs) that represent the most significant patterns in the graph data.

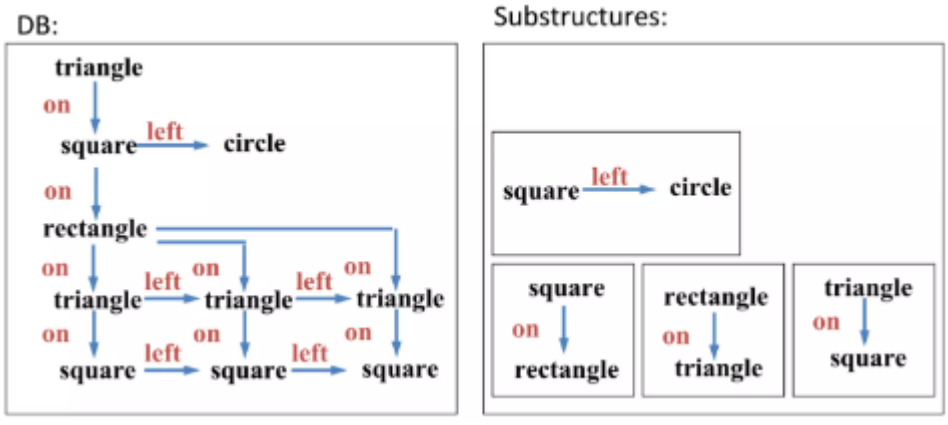
- Subdue uses a beam search strategy, where it maintains a fixed number of best substructures at each step, pruning less promising ones to focus on the most promising candidates. The algorithm stops when no further compression can be achieved, or a predefined number of iterations is reached.

2.4.2 Step-by-step

- Input: A single graph or set of graphs
- Output: List of substructures
- Step-by-step:
  - Step 1: Create substructures for each unique vertex label.



  - Step 2: Extend the best substructure by an edge or an edge and a neighboring vertex.

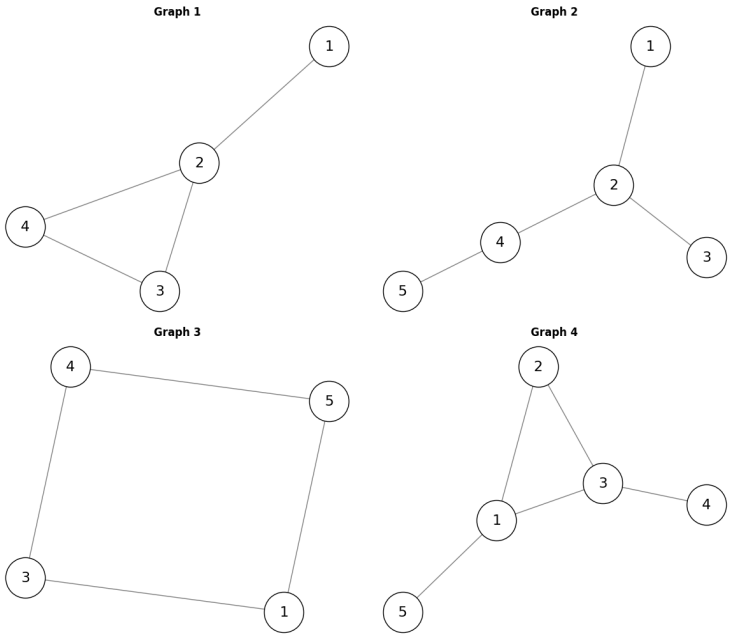


  - Step 3: Keep only the best substructures in the queue (specified by the strip width).
  - Step 4: End when the queue is empty or when the number of detected substructures is greater than or equal to the specified limit.
  - Step 5: Compress the graph and repeat to generate a hierarchical description.
- Pseudo code

Algorithm 6 Subdue(Graph, BeamWidth, MaxBest, MaxSubSize, Limit)

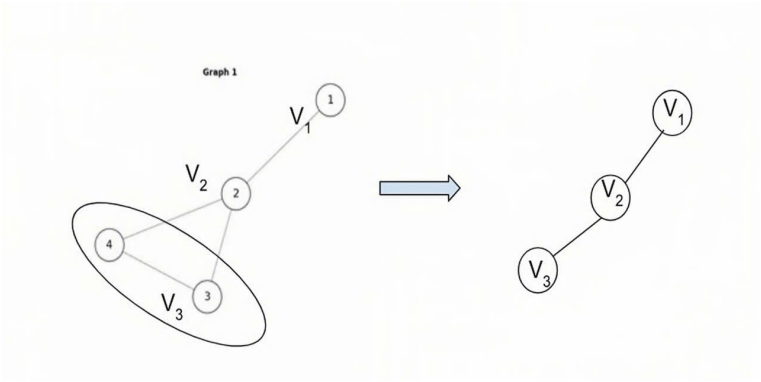
```
ParentList ← Null
ChildList ← Null
BestList ← Null
ProcessedSubs ← 0
Create a substructure from each unique vertex label and its single-vertex instances
Insert the resulting substructures into ParentList
while ProcessedSubs ≤ Limit and ParentList is not empty do
    while ParentList is not empty do
        Parent ← RemoveHead(ParentList)
        Extend each instance of Parent in all possible ways
        Group the extended instances into Child substructures
        for each Child do
            if SizeOf(Child) < MaxSubSize then
                Evaluate the Child
                Insert Child in ChildList in order by value
                if BeamWidth < Length(ChildList) then
                    Destroy substructure at end of ChildList
                end if
            end if
        end for
        Increment ProcessedSubs
        Insert Parent in BestList in order by value
        if MaxBest < Length(BestList) then
            Destroy substructure at end of BestList
        end if
    end while
    Switch ParentList and ChildList
end while
return BestList
```

2.4.3 Example of the algorithm Subdue

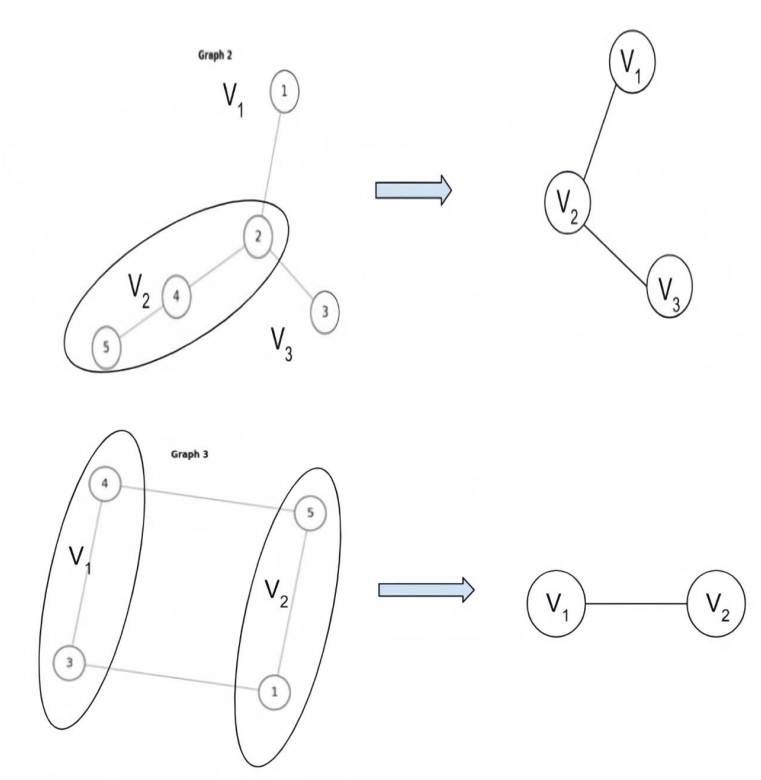


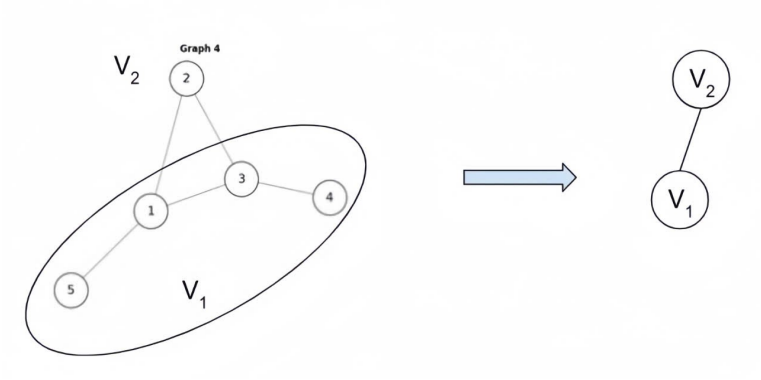
- Graph 1: $V = 1, 2, 3, 4, E = (1, 2), (1, 3), (2, 3), (2, 4)$
- Assumption: The labels of the vertices are as follows: Vertice 1: V1, Vertice 2: V2, Vertice : V3, and Vertice 4: V4.

- Edges are labeled as E:
  - Edge(1,2): E12
  - Edge(1,3): E13
  - Edge(2,3): E23
  - Edge(2,4): E24
- Start from individual vertices: V1, V2, V3, V4
- Then, expand them with edges: (V1,V2), (V1,V3), (V2,V3), (V2,V4)
- Subgraph consisting of 3 vertices and 2 edges: (V1,V2), (V2,V3)



- Similarly, we obtain the results of the following graphs:





2.4.4 Advantages and Disadvantages

- Advantages:
  - Pattern Discovery in Graphs:Subdue is highly effective at discovering significant and repetitive substructures within graph-based data, making it useful in domains like bioinformatics, social networks, and chemical informatics.
  - The use of Minimum Description Length (MDL) as a criterion ensures that the algorithm identifies substructures that provide the most compact representation of the data.
  - By using a beam search strategy and pruning less promising candidates, Subdue can scale to relatively large graphs. The beam search helps manage the complexity by focusing on the most promising substructures at each step.
- Disadvantages:
  - Despite the use of beam search, the process of expanding and evaluating substructures can be computationally intensive, especially for very large or dense graphs. The complexity increases with the size of the graph and the beam width.
  - Subdue relies on heuristic search methods, which means it may sometimes converge to local optima rather than finding the globally best solution. The quality of the discovered substructures can depend on the initial choices and the parameters used in the search.
  - The performance of Subdue can be sensitive to the choice of parameters, such as the beam width and stopping criteria.
  - It is not suitable for non-graph data without significant preprocessing.

3) Graph Partition

3.1 Introduction

- A graph is a mathematical structure consisting of nodes (or vertices) connected by edges. In many real-world scenarios, graphs can be large and complex, making them difficult to analyze or process.
- Graph partitioning is a fundamental technique in graph theory and computer science, helping divide a graph into smaller, more manageable subgraphs, each containing a subset of the original graph's nodes, to minimize the number of edges between different subgraphs (known as the cut size).
- This process is crucial in various applications, from optimizing parallel computing tasks to analyzing large-scale networks like social networks, biological networks, and communication systems.
- Objective of Graph Partitioning:
  - The primary goal is to divide the graph into k partitions, where k is a predefined number, such that the number of edges connecting nodes in different partitions (cut edges) is minimized.
  - Additionally, the partitions should be balanced, meaning that the number of nodes or the computational load in each partition is roughly equal.
- Key techniques: BFS, Kerninghan-Lin (KL) Algorithm, Fiduccia-Mattheyses (FM) Algorithm, Spectral Bisection, K-medois Algorithm
- Applications:
  - Parallel Computing: Efficiently distributing tasks across multiple processors by ensuring that inter-processor communication is minimized.
  - VLSI Design: Partitioning circuits into smaller, manageable sub-circuits to optimize layout and performance.
  - Community Detection: Identifying clusters or communities within social networks by partitioning the graph into groups of densely connected nodes.

3.2 Kernighan-Lin (KL) Algorithm

- The Kernighan-Lin (KL) algorithm *[4]*, developed by Brian W. Kernighan and Shen Lin in 1970, is an important heuristic method in graph partitioning. The goal of the algorithm is to find a way to divide a graph into two parts such that the number of edges connecting the two parts is minimized, while also ensuring that the sizes of the two parts are relatively balanced.
- The algorithm has important practical application in the layout of digital circuits and components in the electronic design automation of VLSI

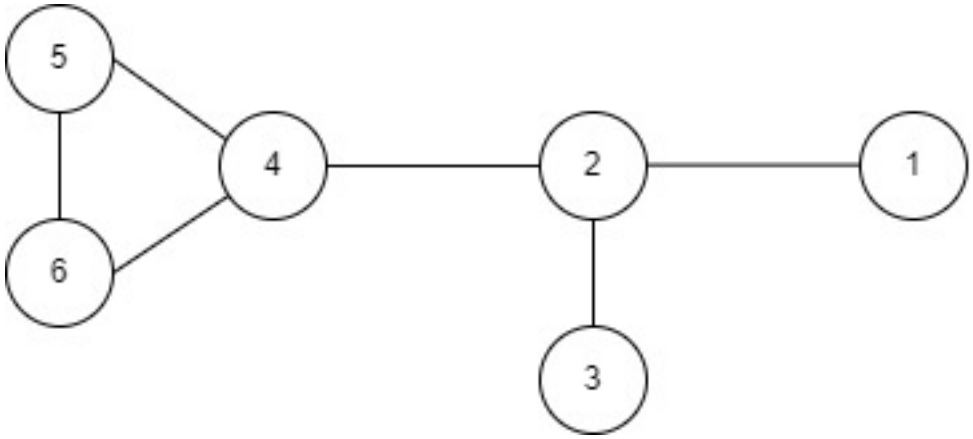
- Description
  - The algorithm partitions an undirected graph $G = (V, E)$ into two disjoint subsets A and B, where V is the set of vertices and E is the set of edges.
  - If the edges have numerical weights, the algorithm aims to minimize the total weight T of the subset of edges that cross from A to B. If the graph is unweighted, then instead the goal is to minimize the number of crossing edges; this is equivalent to assigning weight one to each edge
  - It maintains and improves a partition, in each pass using a greedy algorithm to pair up vertices of A with vertices of B, so that moving the paired vertices from one side of the partition to the other will improve the partition
  - For each $a \in A$,
    - let I_a be the internal cost of a, that is, the sum of the costs of edges between a and other nodes in A
    - let E_a be the external cost of a, that is, the sum of the costs of edges between a and nodes in B. Similarly, define I_b and E_b for each $b \in B$.
  - Let $D_s = E_s - I_s$ be the difference between the external and internal costs of s.
  - If a and b are interchanged, then the cost reduction is $T_{old} - T_{new} = D_a + D_b - 2c_{a,b}$ where $c_{a,b}$ is the cost of the possible edge between a and b.

- Pseudo code

Algorithm 7 Kernighan-Lin Algorithm

```
1: function KERNIGHAN-LIN(G(V, E))
2:     Determine a balanced initial partition of the nodes into sets A and B
3:     repeat
4:         Compute D values for all a ∈ A and b ∈ B
5:         Let gv, av, and bv be empty lists
6:         for n = 1 to |V|/2 do
7:             Find a ∈ A and b ∈ B such that g = D[a] + D[b] − 2 × c(a, b) is maximal
8:             Remove a and b from further consideration in this pass
9:             Add g to gv, a to av, and b to bv
10:            Update D values for the elements of A = A \ {a} and B = B \ {b}
11:        end for
12:        Find k which maximizes g_max, the sum of gv[1], ..., gv[k]
13:        if g_max > 0 then
14:            Exchange av[1], av[2], ..., av[k] with bv[1], bv[2], ..., bv[k]
15:        end if
16:    until g_max ≤ 0
17:    return G(V, E)
18: end function
```

- For example:



– **Step 1:** Initial Partition: $A = \{2, 3, 4\}$ and $B = \{1, 5, 6\}$

– **Iteration 1:**

∗ **Step 2:** Compute D

Partition A	E_a	I_a	**Compute D**	**Partition B**	E_b	I_b	**Compute D2**
2	1	2	-1	1	1	0	1
3	0	1	-1	5	1	1	0
4	2	1	1	6	1	1	0

∗ **Step 3:** Computing gain

Possible pairs	$C_{a,b}$	Gain
$G_{2,1}$	1	-2
$G_{2,5}$	0	-1
$G_{2,6}$	0	-1
$G_{3,1}$	0	0
$G_{3,5}$	0	-1
$G_{3,6}$	0	-1
$G_{4,1}$	1	2
$G_{4,5}$	0	-1
$G_{4,6}$	1	-1

Largest G values: $G_{4,1} = +2$
$A' = A - \{4\} = \{2, 3\}$
$B' = B - \{1\} = \{5, 6\}$

– **Iteration 2:**

∗ Update partitions by swapping 4 and 1, $A' = \{2, 3, 1\}$ and $B' = \{4, 5, 6\}$

∗ **Step 4:** Compute D

Partition A	E_a	I_a	**Compute D**	**Partition B**	E_b	I_b	**Compute D2**
2	1	2	-1	4	1	2	-1
3	0	2	-2	5	0	2	-2
1	0	1	-1	6	0	2	-2

* **Step 5:** Computing gain

Possible pairs	$C_{a,b}$	Gain
$G_{2,5}$	0	-3
$G_{2,6}$	0	-3
$G_{3,5}$	0	-4
$G_{3,6}$	0	-4

Largest G values: $G_{2,5} = -3$
$A' = A' - \{2\} = \{3\}$
$B' = B' - \{5\} = \{6\}$

– **Iteration 3:**

* Update partitions by swapping 2 and 5, $A' = \{5, 3, 1\}$ and $B' = \{4, 2, 6\}$
* **Step 6:** Compute D

Partition A	E_a	I_a	**Compute D**	**Partition B**	E_b	I_b	**Compute D2**
5	2	0	2	4	1	1	0
3	1	0	1	2	2	1	1
1	1	0	1	6	1	1	0

* **Step 7:** Computing gain

Possible pairs	$C_{a,b}$	Gain
$G_{3,6}$	0	1

Largest G values: $G_{3,6} = 1$
$A' = A' - \{2\} = \emptyset$
$B' = B' - \{5\} = \emptyset$

– **Step 8:** Determine K: $G_{4,1} + G_{2,5} + G_{3,6} = 2 - 3 + 1 = 0$

– **Final Partitions:**

* **Partition A:** $\{2, 3, 1\}$
* **Partition B:** $\{4, 5, 6\}$

- Advantages:

– It has a polynomial time complexity, specifically $O(n^2 \log n)$, making it relatively efficient for moderately sized graphs.

– KL iteratively improves the quality of the partition by exchanging pairs of nodes between partitions to minimize the edge cut, leading to better results compared to initial random partitions.

- The algorithm can be adapted to different cost functions, allowing it to be tailored for specific applications or problem constraints.

- Disadvantages:
  - Results are random as the algorithm starts with a random partition.
  - Only two partitions are created.
  - The KL algorithm may get stuck in local optima, as it is a greedy algorithm that makes local improvements. It might not find the global minimum edge cut, especially for complex graphs.
  - While efficient for medium-sized graphs, the $O(n^2 \log n)$ complexity can be computationally expensive for very large graphs, especially if multiple iterations or refinements are required.
  - As the graph size increases, the performance and scalability of the KL algorithm may degrade, especially in scenarios where the graph is sparse or has a large number of vertices.

3.3 Spectral Bisection

- Spectral bisection [3] is a graph partitioning technique that uses the eigenvalues and eigenvectors of the graph's Laplacian matrix to divide the graph into two approximately equal-sized parts.

- Background
  - Let $G = (V, E)$ be an undirected, unweighted graph without loops or multiple edges from one node to another, on $|V| = n$ vertices. Let $A = A(G)$ be an n x n adjacency matrix relevant to G with one row and column for each node, $a_{u,v}$ equal to one if $(u, v) \in E$ and zero otherwise.
  - The Laplacian matrix $L(G)$ of G is an $n \times n$ symmetric matrix with one row and column for each node defined by

$$L_{ij}(G) = \begin{cases} d_i & \text{if } i = j \\ -1 & \text{if } i \neq j \text{ and } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

    for $i, j = 1, ..., n$, where d_i is the vertex degree of node i. Since the Laplacian matrix is symmetric, all eigenvalues are real. The matrices $L(G)$ and $A(G)$ are related via $L = D - A$, D is a diagonal matrix where d_i is the vertex degree of node i.
  - Let the eigenvalues of $L(G)$ be ordered $\lambda_0 = 0 \leq \lambda_1 \leq \ldots \leq \lambda_{n-1}$. An eigenvector corresponding to λ_0 is vector of all ones. The multiplicity of λ_0 is equal to the number of connected components of the graph. The second smallest eigenvalue λ_1 is greater then zero if G is connected.
  - Fiedler also investigated graph-theoretical properties of the eigenvector corresponding to λ_1, called second eigenvector. The coordinates of this eigenvector are assigned to the vertices of

G in a natural way and can be considered as valuation of the vertices of G. Fiedler called this valuation characteristic valuation of G.

- Description
  - Let $G = (V, E)$ be a connected graph, and let $\vec{v} = (v_0, v_1, \ldots, v_n)$ be a second eigenvector - the Fiedler vector of the Laplacian matrix of G. In the following, we always consider G is connected thus multiplicity of the zero eigenvalue is one and the second eigenvalue is greater than zero.
  - The idea of spectral partitioning is to find a splitting value s such that we can partition the vertices in G concerning the evaluation of the Fiedler vector.

- Pseudo code

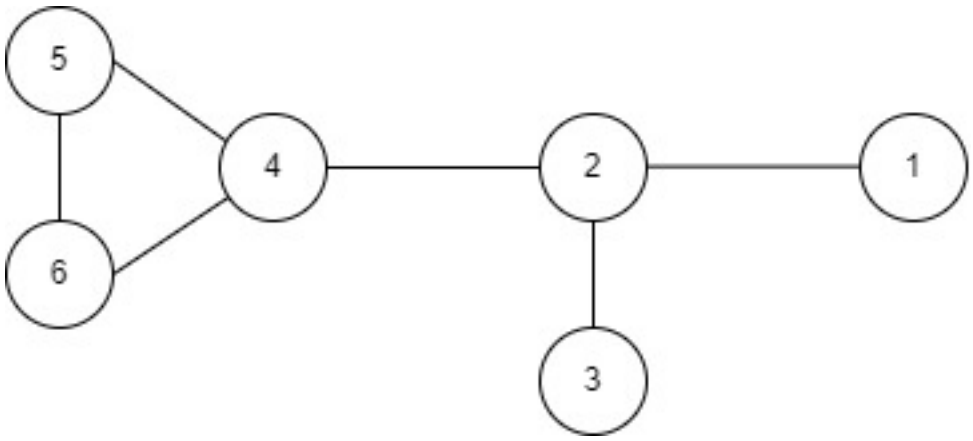
Algorithm 8 Spectral Bisection

Input: A graph $G = (V, E)$
Output: Graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$

Compute the Fiedler eigenvector v
Find the median of v
for each node i in G **do**
    if $v(i) \leq$ median **then**
        Put node i in partition V_1
    else
        Put node i in partition V_2
    end if
end for
if $|V_1| - |V_2| > 1$ **then**
    Move some vertices with components equal to median from V_1 to V_2 to make this difference at most one
end if
Let V_1' be the set of vertices in V_1 adjacent to some vertex in V_2
Let V_2' be the set of vertices in V_2 adjacent to some vertex in V_1
Set up the edge separator E' - the set of edges of G with one point in V_1' and the second in V_2'

Let E_1 be the set of edges with both end vertices in V_1
Let E_2 be the set of edges with both end vertices in V_2
Set up the graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$

- For example:



  - Vertices $V = 1, 2, 3, 4, 5, 6$, with edges $E = (1, 2), (2, 3), (2, 4), (4, 5), (4, 6), (5, 6)$

– **Step 1:** Construct the Laplacian Matrix

$$D = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}, A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

$$L = D - A = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 3 & -1 & -1 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 3 & -1 & -1 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & -1 & 2 \end{pmatrix}$$

– **Step 2:** Find Fiedler eigenvector v

$$eigenvectors = \begin{pmatrix} 0.4082 & 0.4647 & -0.7071 & 0.2887 & 0 & 0.1845 \\ 0.4082 & 0.2610 & 0 & -0.5774 & 0 & -0.6572 \\ 0.4082 & 0.4647 & 0.7071 & 0.2887 & 0 & 0.1845 \\ 0.4082 & -0.2610 & 0 & -0.5774 & 0 & 0.6572 \\ 0.4082 & -0.4647 & 0 & 0.2887 & -0.7071 & -0.1845 \\ 0.4082 & -0.4647 & 0 & 0.2887 & 0.7071 & -0.1845 \end{pmatrix}$$

$$v = \begin{pmatrix} 0.4647 \\ 0.261 \\ 0.4647 \\ -0.26107 \\ -0.4647 \\ -0.4647 \end{pmatrix}$$

– **Step 3: Find median and partition**

* median = 0
* Partition A(value > 0):$\{2, 3, 1\}$
* Partition B:$\{4, 5, 6\}$

- Advantages:

– Spectral bisection provides a global perspective of the graph, as it leverages the eigenvectors

of the Laplacian matrix, particularly the Fiedler vector, which represents the graph's overall structure.

  - Spectral methods can be effective for large and complex graphs, especially when traditional combinatorial methods struggle. The eigenvector-based approach captures essential structural properties, making it scalable for large datasets.
  - The partitions obtained by spectral bisection tend to be smoother and more natural compared to other methods. This is especially useful in applications like image segmentation or clustering, where contiguous regions or clusters are desired.

- Disadvantages:
  - Computing the eigenvectors of the Laplacian matrix can be computationally expensive, especially for large graphs. The cost is $O(n^3)$ for exact methods, though there are faster approximate methods. This can be a bottleneck for very large graphs.
  - The effectiveness of spectral bisection can be highly dependent on the structure of the graph. For example, it may perform poorly on graphs with tightly connected subgraphs or graphs with very irregular structures.
  - The connection between the eigenvectors and the actual graph structure may not be straightforward, making it harder to understand why a particular partition was chosen.
  - The performance of spectral bisection can depend on the gap between the smallest eigenvalues of the Laplacian matrix. If the gap is small, the partitioning quality might degrade, leading to less distinct partitions.

4) Node Embedding

4.1 Introduction

- Node embedding is a technique used in machine learning and network science to represent nodes (or vertices) in a graph as low-dimensional vectors. These vectors capture the structural and relational properties of the nodes in a way that makes them suitable for use in various machine-learning tasks, such as node classification, link prediction, clustering, and visualization.
- Traditional machine learning algorithms typically require inputs in a fixed-size vector format. However, nodes in a graph do not naturally have a vector representation, and the number of nodes can vary across different graphs. Node embedding algorithms solve this problem by learning a mapping from nodes to a continuous vector space, where each node is represented by a fixed-size vector.
- Goals of Node Embedding

  - Preserve Network Structure: The embedding should capture the local and/or global structure of the graph. For instance, nodes that are closely connected in the graph should have similar vector representations.
  - Dimensionality Reduction: By mapping nodes to a lower-dimensional space, node embeddings reduce the complexity of the data while preserving essential properties.
  - Applicability to Downstream Tasks: The learned embeddings should be useful for tasks like node classification (predicting labels of nodes), link prediction (predicting the existence of edges between nodes), and clustering (grouping similar nodes).

- Methods for Node Embedding:
  - Random Walk-Based Methods:DeepWalk, Node2vec
    - **DeepWalk**: Generates node embeddings by simulating random walks on the graph and treating the sequence of nodes visited as a "sentence" in a language model, similar to Word2vec.
    - **Node2vec**: An extension of DeepWalk that allows for flexible random walks, balancing between breadth-first and depth-first search strategies.
  - Matrix Factorization-Based Methods: These methods factorize a matrix that captures the node-node similarity, such as the adjacency matrix or Laplacian matrix of the graph, to obtain embeddings.
  - Graph Neural Networks (GNNs):
    - GNNs are a more recent and powerful approach to node embedding, where neural networks are used to aggregate and transform information from a node's neighborhood to produce its embedding.
    - Edge/Link Prediction-Based Methods: Methods like LINE (Large-scale Information Network Embedding) preserve first-order (direct links) and second-order (shared neighbors) proximities in the embeddings.

- Applications:
  - Social Network Analysis: Node embeddings can be used to identify communities, recommend friends, or detect anomalies.
  - Bioinformatics: Embeddings can represent proteins or genes in a network, helping in the prediction of protein-protein interactions or gene-disease
  - Recommendation Systems: In user-item interaction graphs, node embeddings can help recommend items to users based on their embedding similarities.

4.2 Deepwalk

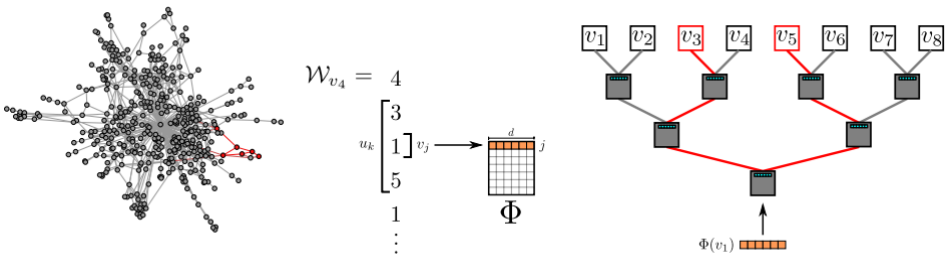
4.2.1 Description

- The core problem addressed by DeepWalk *[8]* is to learn representations (embeddings) of nodes in a graph such that nodes that are "close" in the graph are also close in the embedding space. These embeddings can then be used for tasks like node classification, clustering, and link prediction.
- DeepWalk introduces the idea of using random walks combined with the Skip-Gram model (from natural language processing) to learn node embeddings. This approach is novel in that it treats walks on the graph as sentences in a language model, enabling the application of word embedding techniques to graph data.
- DeepWalk is an unsupervised method, meaning it does not require labeled data to learn the embeddings, making it applicable to a wide variety of real-world graphs where labeled data might be scarce or unavailable.

4.2.2 Workflow

How the DeepWalk Algorithm works:

- The algorithm generates random walks of fixed length starting from each node in the graph. A random walk is a sequence of nodes, where each node in the sequence is selected randomly from the neighbors of the previous node.
- Skip-Gram Model
  - The random walks are treated as sentences, where each node is a "word". The Skip-Gram model, originally used in word2vec for learning word embeddings, is then applied to these sequences.
  - The Skip-Gram model learns to predict the context (neighboring nodes) of a node within a walk, thereby capturing the structural relationships in the graph.
- Optimization
  - The learning objective is to maximize the log probability of observing a node's context in the random walk, given the node's representation.
  - To make this optimization efficient, DeepWalk uses hierarchical softmax, a technique that speeds up the training process by approximating the softmax function over the vocabulary of nodes.
- Embedding Representation: After training, each node in the graph is represented as a d-dimensional vector (where d is the embedding size), capturing its structural properties within the graph.



(a) Random walk generation. (b) Representation mapping. (c) Hierarchical Softmax.

- Pseudo code:

Algorithm 9 DeepWalk(G, w, d, γ, t)

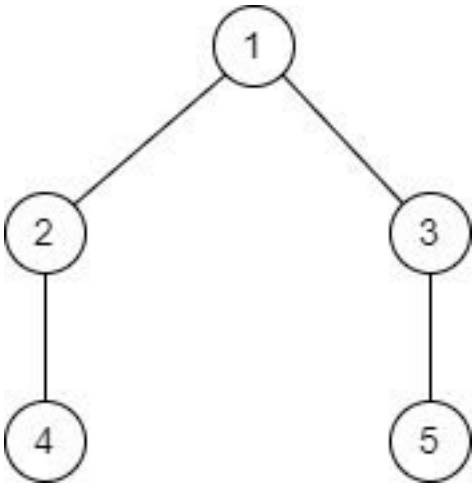
Input:graph $G(V, E)$, window size w, embedding size d, walks per vertex γ, walk length t
Output: matrix of vertex representations $\Phi \in \mathcal{R}^{|V| \times d}$
Initialization: Sample Φ from $\mathcal{U}^{|V| \times d}$
Build a binary tree T from V
for $i = 0$ to γ **do**
    $\mathcal{O} = \text{Shuffle}(V)$
    for each $v_i \in \mathcal{O}$ **do**
        $W_{v_i} = \text{RandomWalk}(G, v_i, t)$
        SkipGram(Φ, W_{v_i}, w)
    end for
end for

Algorithm 10 SkipGram(Φ, W_{v_i}, w)

for each $v_j \in W_{v_i}$ **do**
    for each $u_k \in W_{v_i}[j - w : j + w]$ **do**
        $J(\Phi) = -\log \Pr(u_k \mid \Phi(v_j))$
        $\Phi = \Phi - \alpha * \frac{\partial J}{\partial \Phi}$
    end for
end for

4.2.3 Example of Deepwalk

Given Graph G having $V = 1, 2, 3, 4, 5, E = (1, 2), (1, 3), (2, 4), (3, 5)$



- **Step 1: Generate Random Walks**: Suppose we perform 2 random walks from each node with a length of 4 steps. For example:
  - From node 1, we might obtain sequences like: (1, 2, 4, 2), (1, 3, 5, 3)
  - From node 2, we might obtain sequences like: (2, 1, 3, 1), (2, 4, 2, 1)
  - From node 3, we might obtain sequences like: (3, 1, 2, 4), (3, 5, 3, 1)
  - From node 4, we might obtain sequences like: (4, 2, 1, 3), (4, 2, 4, 2)
  - From node 5, we might obtain sequences like: (5, 3, 1, 2), (5, 3, 5, 3)
- **Step 2: Construct Context-Target Pairs from Random Walks**: From the sequences obtained in the previous step, we identify context-target pairs using a sliding window. Suppose the sliding window size is 2, we can obtain pairs like:
  - From the sequence $1 \to 2 \to 4 \to 2$:

    Pairs: $(1, 2), (1, 4)$

    Pairs: $(2, 1), (2, 4)$

    Pairs: $(4, 2), (4, 1)$

    Pairs: $(2, 4), (2, 1)$
  - From the sequence $1 \to 3 \to 5 \to 3$:

    Pairs: $(1, 3), (1, 5)$

    Pairs: $(3, 1), (3, 5)$

    Pairs: $(5, 3), (5, 1)$

    Pairs: $(3, 5), (3, 1)$
  - These pairs will be used to update embedding vectors similarly to how the Skip-gram model is used in Node2Vec.
- **Step 3: Train the Model**: Apply the Skip-gram model with the context-target pairs obtained from the previous step. This is similar to Node2Vec, where the embedding vectors $\mathbf{z}_v$ of the nodes are updated through gradient descent.

– **Specific Calculations** Suppose the initial vectors are:

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \end{bmatrix} = \begin{bmatrix} 0.2 & -0.4 \\ -0.1 & 0.3 \\ 0.4 & 0.1 \\ 0.3 & -0.2 \\ -0.2 & 0.5 \end{bmatrix}$$

– **Calculate the Dot Product of Vectors** For the pair $(1, 2)$:

$$\mathbf{z}_1 \cdot \mathbf{z}_2 = (0.2 \times -0.1) + (-0.4 \times 0.3) = -0.02 - 0.12 = -0.14$$

– **Calculate the Probability for the Pair (1, 2)** Apply the softmax function to compute the probability.

$$p(2 \mid 1) = \frac{\exp(\mathbf{z}_1 \cdot \mathbf{z}_2)}{\sum_k \exp(\mathbf{z}_1 \cdot \mathbf{z}_k)} \approx 0.4$$

, where $\text{score}_{1,k}$ are scores computed from other context-target pairs.

– **Calculate the Loss Function** The loss function for the Skip-gram model can be written as:

$$L = -\log p(2 \mid 1) - \log 0.4 \approx 0.916$$

– **Compute Gradient and Update Embedding Vectors** The gradient of the loss function with respect to vectors $\mathbf{z}_1$ and $\mathbf{z}_2$ is computed as follows:

$$\frac{\partial L}{\partial \mathbf{z}_1} = -(p(2 \mid 1) - 1) \cdot \mathbf{z}_2 == -(0.4 - 1) \cdot \mathbf{z}_2 = 0.6 \cdot [-0.1, 0.3] = [-0.06, 0.18]$$

$$\frac{\partial L}{\partial \mathbf{z}_2} = -(p(2 \mid 1) - 1) \cdot \mathbf{z}_1 = -(0.4 - 1) \cdot \mathbf{z}_1 = 0.6 \cdot [0.2, -0.4] = [0.12, -0.24]$$

– With $\eta = 0.1$ as the learning rate,

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \end{bmatrix} \approx \begin{bmatrix} 0.2006 & -0.4018 \\ -0.1012 & 0.3024 \\ 0.4006 & 0.0985 \\ 0.3004 & -0.2012 \\ -0.2012 & 0.4997 \end{bmatrix}$$

4.3 Node2vec

4.3.1 Description

- Node2Vec is a node embedding algorithm that computes vector representations of nodes based on random walks in the graph.
- The neighborhood is sampled through random walks. Using several samples of random neighborhood walks, the algorithm trains a single hidden-layer neural network.
- The neural network is trained to predict the probability of a node appearing in a walk based on the presence of another node.
- In Node2Vec, the sampling of random walks is not purely random but is influenced by two additional hyperparameters that control the bias in the random walk sampling process:
  - p - return parameter
  - q - in-out parameter

4.3.2 Mathematical basis

- Let $G = (V, E)$ be a given network. Our analysis is general and applies to any (un)directed, (un)weighted network.
- Let $f : V \to \mathbb{R}^d$ be the mapping function from nodes to feature representations we aim to learn for a downstream prediction task. Here d is a parameter specifying the number of dimensions of our feature representation.
- Equivalently, f is a matrix of size $|V| \times d$ parameters.
- For every source node $u \in V$, we define $N_S(u) \subseteq V$ as a network neighborhood of node u generated through a neighborhood sampling strategy S.
- We proceed by extending the Skip-gram architecture to networks.
- We seek to optimize the following objective function, which maximizes the log-probability of observing a network neighborhood $N_S(u)$ for a node u conditioned on its feature representation, given by f:

$$\max_f \quad \sum_{u \in V} \log Pr(N_S(u)|f(u)).$$

- Conditional independence:

$$Pr(N_S(u)|f(u)) = \prod_{n_i \in N_S(u)} Pr(n_i|f(u)).$$

- Symmetry in feature space:

$$Pr(n_i|f(u)) = \frac{\exp(f(n_i) \cdot f(u))}{\sum_{v \in V} \exp(f(v) \cdot f(u))}.$$

- With the above assumptions, the objective in Eq. 1 simplifies to:

$$\max_f \sum_{u \in V} \left[-\log Z_u + \sum_{n_i \in N_S(u)} f(n_i) \cdot f(u) \right].$$

- The per-node partition function, $Z_u = \sum_{v \in V} \exp(f(u) \cdot f(v))$, is expensive to compute for large networks and we approximate it using negative sampling.

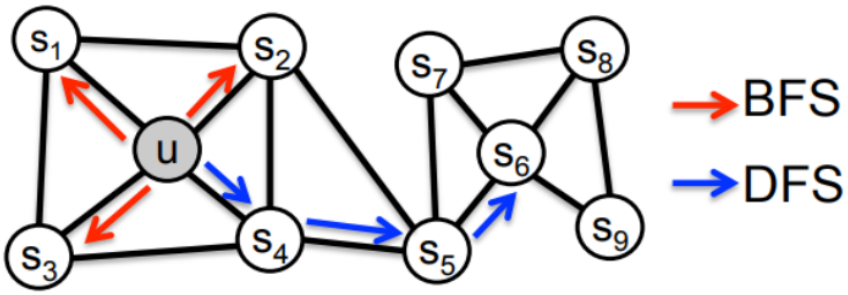


Figure 1: BFS and DFS search strategies from node u ($k = 3$).

4.3.3 Techniques used

- **Random Walks Formally**: Given a source node u, we simulate a random walk of fixed length l. Let c_i denote the i-th node in the walk, starting with $c_0 = u$. Nodes c_i are generated by the following distribution:

$$P(c_i = x \mid c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z} & \text{if } (v,x) \in E \\ 0 & \text{otherwise} \end{cases}$$

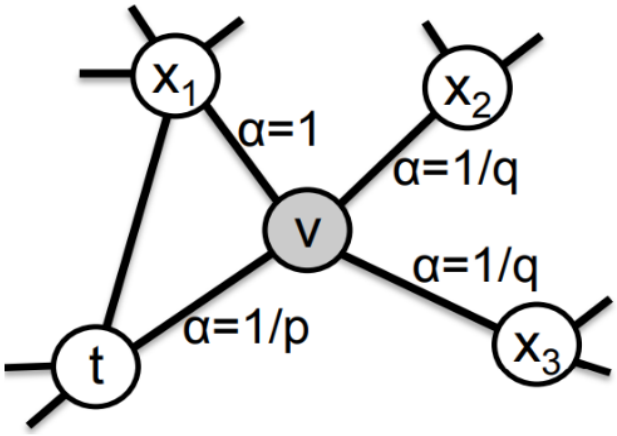
  where π_{vx} is the unnormalized transition probability between nodes v and x, and Z is the normalizing constant.

- **Search Bias** α: The unnormalized transition probability is:

$$\pi_{vx} = \alpha_{pq}(t,x) \times w_{vx},$$

where $\alpha_{pq}(t,x)$ depends on the shortest path distance d_{tx} between nodes t and x:

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0, \\ 1 & \text{if } d_{tx} = 1, \\ \frac{1}{q} & \text{if } d_{tx} = 2. \end{cases}$$



- **Parameters p and q**:
  - **Return Parameter p**: Parameter p controls the likelihood of immediately revisiting a node in the walk. Setting it to a high value ($> \max(q, 1)$) ensures that we are less likely to sample an already visited node in the following two steps (unless the next node in the walk has no other neighbor). This strategy encourages moderate exploration and avoids 2-hop redundancy in sampling. On the other hand, if p is low ($< \min(q, 1)$), it would lead the walk to backtrack a step and this would keep the walk "local," close to the starting node u.
  - **In-out Parameter q**: Parameter q allows the search to differentiate between "inward" and "outward" nodes. If $q > 1$, the random walk is biased towards nodes close to node t. Such walks obtain a local view of the underlying graph with respect to the start node in the walk and approximate BFS behavior in the sense that our samples consist of nodes within a small locality. In contrast, if $q < 1$, the walk is more inclined to visit nodes which are further away from the node t.

4.3.4 Pseudo Code

Algorithm 11 The `node2vec` algorithm.

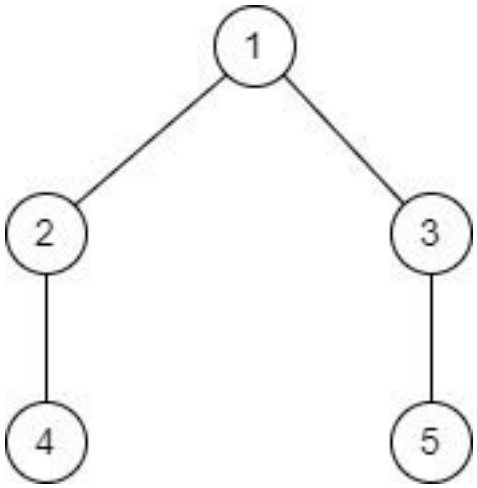
procedure LEARNFEATURES($G = (V, E, W)$, d, r, l, k, p, q)
  $\pi \leftarrow$ PreprocessModifiedWeights(G, p, q)
  $G' = (V, E, \pi)$
  Initialize *walks* to Empty
  for iter = 1 to r **do**
    for all nodes $u \in V$ **do**
      $walk \leftarrow$ node2vecWalk(G', u, l)
      Append *walk* to *walks*
    end for
  end for
  $f \leftarrow$ StochasticGradientDescent($k, d, walks$)
  return f
end procedure

Algorithm 12 `node2vecWalk`

procedure NODE2VECWALK($G' = (V, E, \pi)$, u, l)
  Initialize *walk* to [u]
  for walk_iter = 1 to l **do**
    $curr \leftarrow$ walk[-1]
    $V_{curr} =$ GetNeighbors($curr, G'$)
    $s \leftarrow$ AliasSample(V_{curr}, π)
    Append s to *walk*
  end for
  return *walk*
end procedure

4.3.5 Example

Given Graph G having $V = 1, 2, 3, 4, 5, E = (1, 2), (1, 3), (2, 4), (3, 5)$



- **Assumed Parameters:**
  - $p = 1$: The probability of returning to the previous node is medium.
  - $q = 2$: The probability of moving farther from the current node is higher.
  - $l = 3$: The length of each random walk is 3.

- $r = 2$: The number of random walks to perform from each node is 2.
- Embedding dimension $d = 2$.

- **Step 1: Perform Random Walks**
  - Starting from node 1:$(1, 2, 4, 2)$, $(1, 3, 5, 3)$
  - Starting from node 2:$(2, 4, 2, 1)$, $(2, 1, 3, 5)$

- **Step 2: Generate (word, context) Pairs**

$$(1, 2), (2, 4), (4, 2)$$
$$(1, 3), (3, 5), (5, 3)$$
$$(2, 4), (4, 2), (2, 1)$$
$$(2, 1), (1, 3), (3, 5)$$

- **Step 3: Train the Skip-gram Model**. Assume the initial embedding vectors are as follows:

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \end{bmatrix} = \begin{bmatrix} 0.2 & -0.4 \\ -0.1 & 0.3 \\ 0.4 & 0.1 \\ 0.3 & -0.2 \\ -0.2 & 0.5 \end{bmatrix}$$

- **Example: Updating Vectors for the Pair (1,2)**
  - **Calculate the Dot Product**

$$\mathbf{z}_1 \cdot \mathbf{z}_2 = (0.2 \times -0.1) + (-0.4 \times 0.3) = -0.02 - 0.12 = -0.14$$

  - **Compute the Softmax Probability**

$$P(2 \mid 1) = \frac{\exp(z_1 \cdot z_2)}{\sum_{k \in V} \exp(z_1 \cdot z_k)} = \frac{\exp(-0.14)}{\exp(-0.14) + \text{other terms}} \approx 0.226$$

  - **Update Vectors** Perform Gradient Descent with learning rate $\alpha = 0.1$.

$$\mathbf{z}_1 \leftarrow \mathbf{z}_1 - \alpha \frac{\partial \text{Loss}}{\partial \mathbf{z}_1}$$

$$\mathbf{z}_2 \leftarrow \mathbf{z}_2 - \alpha \frac{\partial \text{Loss}}{\partial \mathbf{z}_2}$$

– **Final Embedding Vectors**. After optimizing with all context pairs, we obtain new embedding vectors for the nodes. The final embedding vectors for the nodes could be:

$$tmp = z1 * \begin{bmatrix} z_2 \\ z_3 \\ z_4 \\ z_5 \end{bmatrix} = \begin{bmatrix} -0.14 \\ 0.04 \\ 0.14 \\ -0.24 \end{bmatrix}, \exp(tmp) \approx \begin{bmatrix} 0.8694 \\ 1.0408 \\ 1.1503 \\ 0.7866 \end{bmatrix}, \sum \exp(tmp) = 3.8471$$

– Update the Vectors Using Gradient Descent. Assume learning rate $\alpha = 0.1$. For $\mathbf{z}_1$:

$$\mathbf{z}_1 = \mathbf{z}_1 + \alpha \cdot (\mathbf{z}_2 - P(2 \mid 1) \cdot \mathbf{z}_2) == [0.19226, -0.37678]$$

– Similarly, update z2, z3, z4, z5:

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \end{bmatrix} = \begin{bmatrix} 0.19226 & -0.37678 \\ -0.08452 & 0.26904 \\ 0.41458 & 0.07084 \\ 0.293518 & -0.18031 \\ -0.1754 & 0.50535 \end{bmatrix}$$

5) Graph Generation

5.1 Introduction

- Graph Generation involves developing algorithms and models to generate new graphs based on a set of data or specific characteristics. Graphs in this context typically consist of nodes (vertices) and the edges that connect them, representing relationships between entities in a complex system. Graph Generation models can learn from existing data and generate new graphs with similar or specific characteristics.
- Graph generative objectives (graph generator) to create synthetic graphs for simulated research.
- Graph requirements are incurred: as close to reality as possible
- Composite graphs are divided into 5 types:
  - Random graph model
  - Preferential attachment model
  - Optimization-based model
  - Geographical model
  - Internet-specific model

5.2 The Watts-Strogatz model

5.2.1 Description

- The Watts-Strogatz model *[1]* is a foundational concept in network science, introduced by Duncan J. Watts and Steven H. Strogatz in their 1998 paper, "Collective dynamics of 'small-world' networks." It provides a simple yet powerful framework for understanding the small-world phenomenon observed in many real-world networks, such as social networks, the internet, and biological systems. Below is a detailed exploration of the model, including its development, mathematical formulation, properties, and applications.

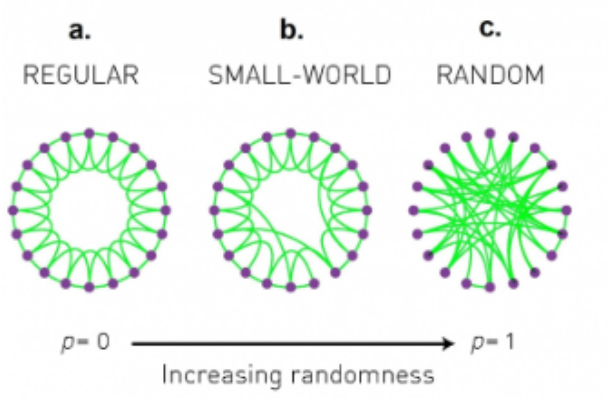
- Real-world networks often exhibit two key properties:

  - High Clustering: Nodes tend to form tightly-knit groups, where most nodes within a group are connected to each other.

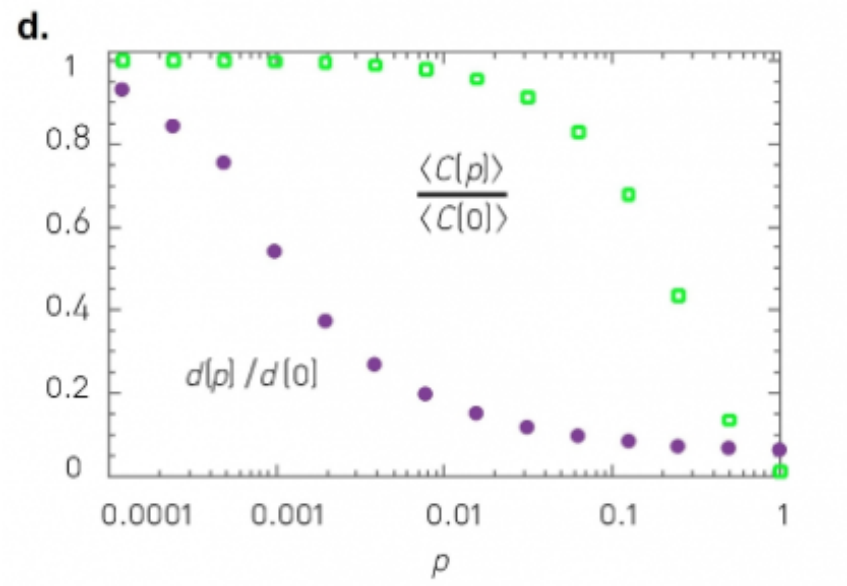
  - Short Average Path Lengths: Despite the clustering, the average distance (in terms of the number of edges) between any two nodes in the network is relatively small, a property referred to as the "small-world" phenomenon.

- Traditional network models at the time, such as regular lattices and random graphs, did not simultaneously exhibit these properties. The Watts-Strogatz model was introduced to bridge this gap.

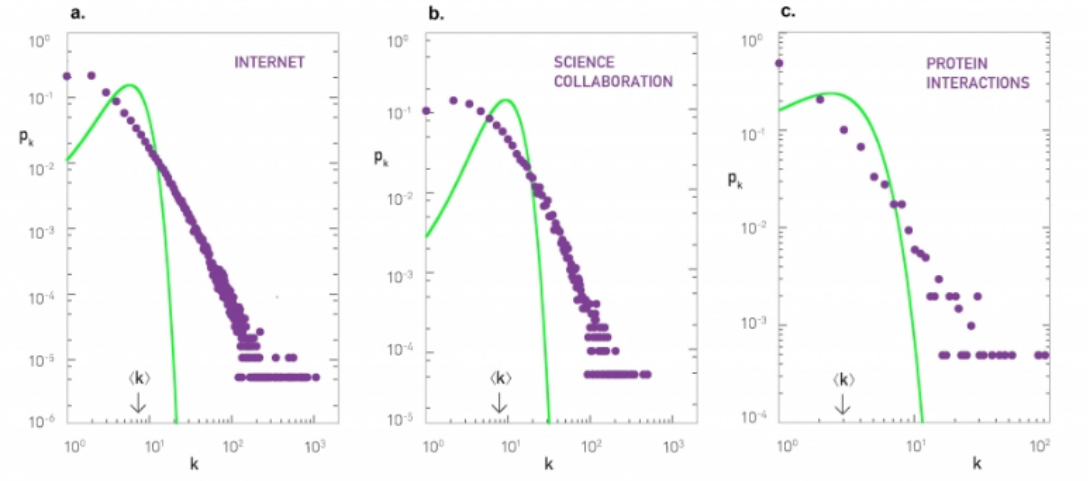
- The Watts-Strogatz model, also known as the small-world model, serves as an interpolation between a regular lattice—characterized by high clustering but lacking the small-world phenomenon—and a random network, which, despite its low clustering, exhibits the small-world property.



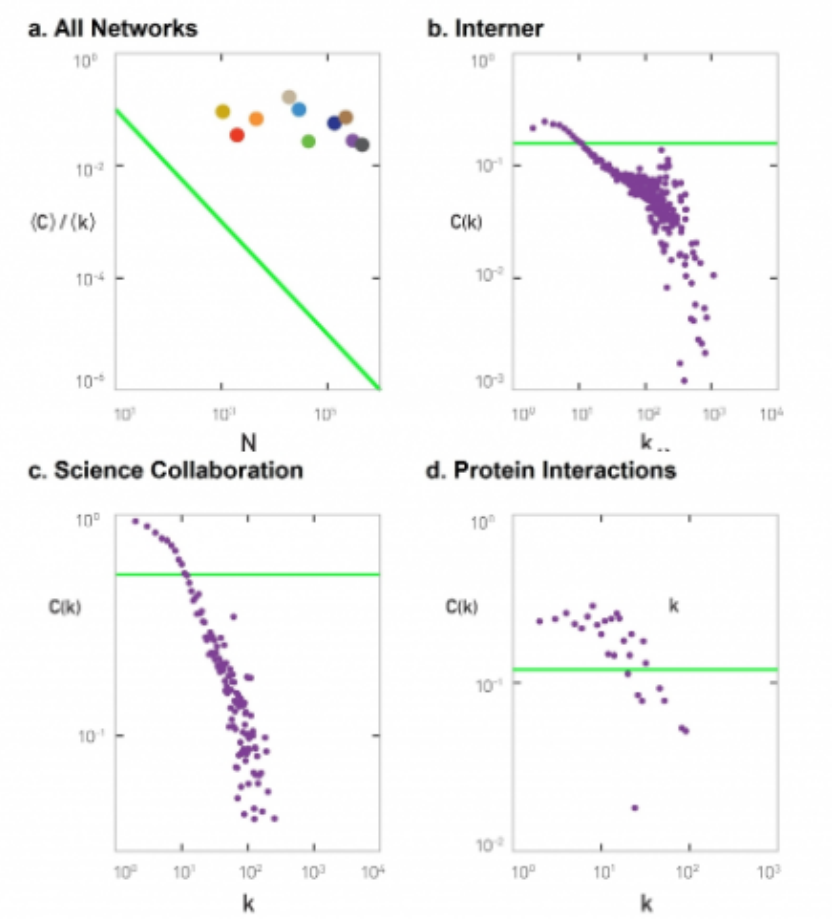
- Numerical simulations demonstrate that within a specific range of rewiring parameters, the model achieves a low average path length while maintaining a high clustering coefficient, thereby replicating the coexistence of high clustering and small-world characteristics.



- As an extension of the random network model, the Watts-Strogatz model predicts a degree distribution similar to a Poisson distribution, which is bounded and thus lacks the presence of high-degree nodes, as illustrated in the accompanying image:



- Additionally, the model predicts a clustering coefficient, C(k), that is independent of the node degree, k, failing to capture the k-dependence observed in the provided images (b, c, d).



- Model Construction:
  - Starting with a ring of nodes, each node is initially connected to its immediate and next neighbors, resulting in an initial average clustering coefficient ‹C› = 3/4 when p = 0.
  - With a probability p, each link is rewired to a randomly chosen node. At small values of p, the network retains high clustering, while the introduction of random long-range links significantly reduces the distances between nodes.
  - When p = 1, all links have been rewired, transforming the network into a random network.
  - The dependence of the average path length, d(p), and the clustering coefficient, ‹C(p)›, on the rewiring parameter, p, is noted. Here, d(p) and ‹C(p)› are normalized by their initial values, d(0) and ‹C(0)›, obtained for a regular lattice (i.e., for p = 0 in (a)). The rapid decrease in d(p) marks the onset of the small-world phenomenon, during which ‹C(p)› remains high. Consequently, within the range of $0.001 < p < 0.1$, short path lengths and high clustering coexist in the network. All graphs presented have N = 1000 and ‹k› = 10.

5.2.2 Limitations and Extensions

While the Watts-Strogatz model captures the small-world phenomenon, it has some limitations:

- **Degree Distribution:** The model predicts a Poisson-like degree distribution, which does not account for the heavy-tailed distributions observed in many real-world networks (e.g., scale-free networks).
- **Lack of Hubs:** High-degree nodes (hubs), which are common in many real-world networks, are absent in the Watts-Strogatz model.
- **K-Independent Clustering:** The clustering coefficient is independent of node degree k, which contrasts with the k-dependent clustering observed in some real-world networks.

5.2.3 Applications

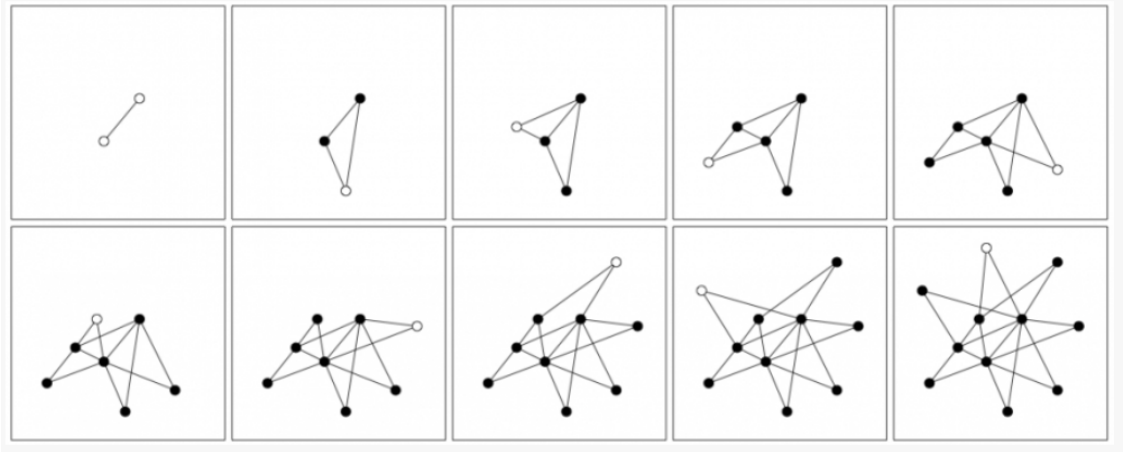
- **Social Networks:** Understanding the spread of information, behaviors, or diseases in social networks.
- **Biological Networks:** Analyzing the structure of neural networks, metabolic networks, and protein-protein interaction networks.
- **Technological Networks:** Examining the robustness and dynamics of communication networks, such as the internet.

5.3 The Barabási-Albert Model

5.3.1 Description

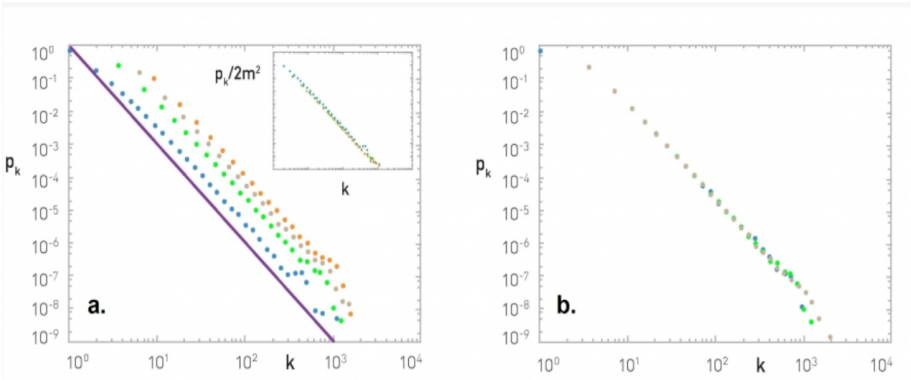
- Growth: Real networks result from a growth process continuously increasing N. In contrast, the random network model assumes that the number of nodes, N, is fixed.
- Preferential Attachment: New nodes tend to link to the more connected nodes in real networks. In contrast, nodes in random networks randomly choose their interaction partners.
- Growth and preferential attachment coexist in real networks has inspired a minimal model called the Barabási-Albert model *[1]*, which can generate scale-free networks
  - Growth: At each timestep, we add a new node with $m(\leq m_0)$ links that connect the new node to m nodes already in the network.
  - Preferential attachment: The probability $\pi(k)$ that a link of the new node connects to node i depends on the degree k_i as

$$\pi(k) = \frac{k_i}{\sum_j k_j}$$



- Degree distribution:

$$p_k = \frac{2m(m+1)}{k(k+1)(k+2)}$$



  - Generating networks with $N = 100,000$ and $m_0 = m = 1$ (blue), 3 (green), 5 (gray), and 7 (orange). The fact that the curves are parallel to each other indicates that γ is independent of m and m_0. The slope of the purple line is -3, corresponding to the predicted degree exponent $\gamma = 3$. $p_k \sim \frac{2m^2}{k^3}$, hence $p_k/2m^2$ should be independent of m. Indeed, by plotting $p_k/2m^2$ vs. k, the data points shown in the main plot collapse into a single curve.
  - The Barabási-Albert model predicts that p_k is independent of N. To test this, we plot p_k for $N = 50,000$ (blue), $100,000$ (green), and $200,000$ (gray), with $m_0 = m = 3$. The obtained p_k values are practically indistinguishable, indicating that the degree distribution is stationary, i.e., independent of time and system size.
  - After experimenting, we figure out that
    - The degree exponent γ is independent of m, a prediction that agrees with the numerical results (Chart a)
    - The power-law degree distribution observed in real networks describes systems of rather different ages and sizes. Hence, an appropriate model should lead to a time-independent

degree distribution. Indeed, according to the degree distribution of the Barabási-Albert model is independent of both t and N. Hence the model predicts the emergence of a stationary scale-free state. Numerical simulations support this prediction, indicating that pk observed for different t (or N) fully overlap (Chart b)

- The analytical calculations predict that the Barabási-Albert model generates a scale-free network with degree exponent $\gamma = 3$. The degree exponent is independent of the m and m_0 parameters. Furthermore, the degree distribution is stationary (i.e. time-invariant), explaining why networks with different history, size, and age develop a similar degree distribution.
- The degree distribution:

$$P_k = \frac{2m^2}{k^3}$$

5.3.2 Steps to Generate a BA Network:

- Initialization: Start with an initial connected network of m_0 nodes.
- Growth: Add new nodes one at a time. Each new node is connected to $m \leq m_0$ existing nodes.
- Preferential Attachment: The probability that a new node connects to an existing node i proportional to the degree k_i of node i.

Algorithm 13 Attach a Node to a Graph

```
function ATTACH(node, G, m)
    Initialize a list V which includes vertices of graph G
    Initialize an empty set E
    while size of E < m do
        v ← a random vertex from V
        Add (node, v) to E
    end while
    Add all edges in E to G
    return G
end function
```

Algorithm 14 BA Graph Generator

```
function BAGraphGenerator(n, m)
    Initialize G as a star graph with m edges
    for i from 1 to n − m − 1 do
        ATTACH(m + i, G, m)
    end for
    return G
end function
```

5.3.3 Limitations and Extensions

- Limitations
  - Assumes new nodes always prefer high-degree nodes, ignoring other factors like geography or interests.

  - Produces networks with lower clustering coefficients than many real-world networks.
  - Cannot easily generate networks with different power-law exponents.
  - Considers only degree for attachment, ignoring other attributes.

- Extensions
  - Reduces node attractiveness over time, modeling relevance decay.
  - Adds layers to represent community structures, increasing clustering.
  - Combines preferential and other attachment rules for more realistic networks.

5.3.4 Applications

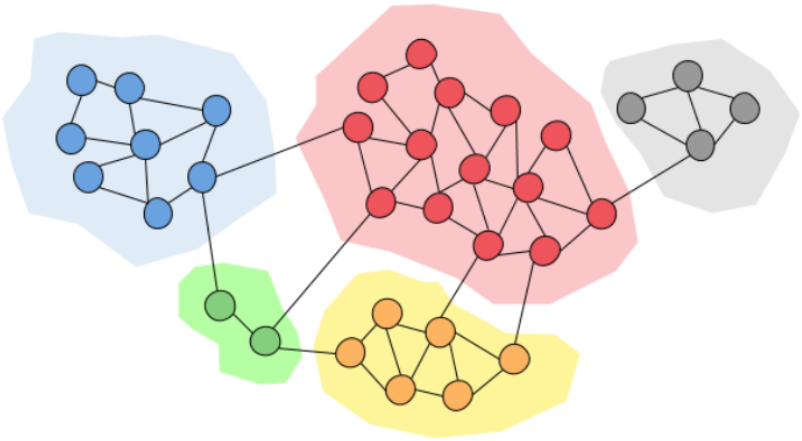
The BA model is used to understand the structure and dynamics of various real-world networks, such as:

- The internet (where websites with more links attract more new links).
- Social networks (where individuals with more connections attract more new connections).
- Biological networks (e.g., metabolic networks, where certain metabolites are more likely to participate in chemical reactions).

6) Community Dectection

6.1 Introduction

- One of the most relevant features of graphs representing real systems is community structure, or clustering, i.e. the organization of vertices in clusters, with many edges joining vertices of the same cluster and comparatively few edges joining vertices of different clusters

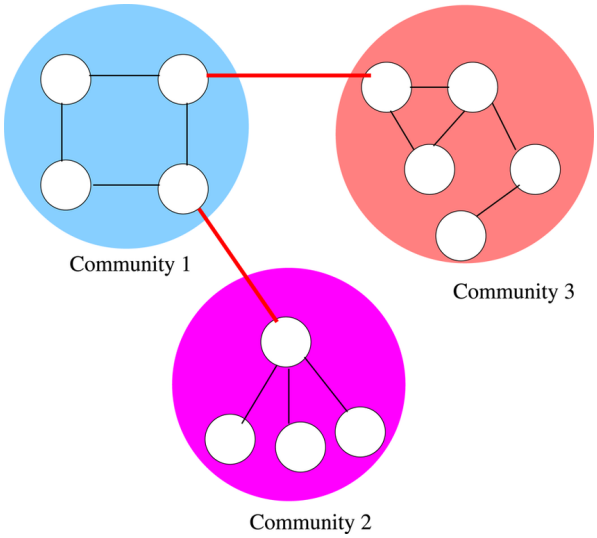


- Community is the set of vertices where each vertex has more internal connections than outward connections. For community optimization, there are the quality and distance measurements, one has 2 more ways of measuring to assess density:
  - Intracluster density: the bigger the better
  - Intercluster density: the smaller the better
- Community Detection is an important technique in graph and network analysis that focuses on identifying subgroups in a graph where nodes are more closely connected than to nodes outside the group.
- Community detection methods
  - Methods based on minimal slices: These methods focus on partitioning a graph by minimizing the number of edges (or the weight of edges) that are cut between different groups or communities. The idea is to find a partitioning of the graph where the number of connections between different communities is minimized, and the number of connections within the same community is maximized.
  - Intermediate-based method: These methods focus on the intermediate steps or processes within the graph to detect communities. They often rely on centrality measures, edge betweenness, or other intermediate properties of the graph.
  - Methods based on RandomWalk: Random walk-based methods utilize the concept of random walks on graphs to detect communities. The intuition is that a random walker starting from a node is likely to stay within the same community for a longer period before exiting to another community.

6.2 Girvan-Newman

6.2.1 Description

- **Community Detection:**

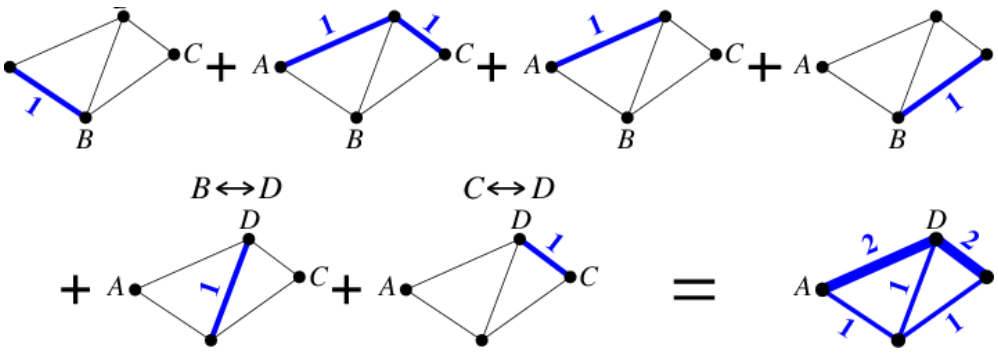


  - Definition: In graph theory, community detection refers to the task of identifying groups of nodes (communities) that have a higher density of edges within the group compared to edges connecting them to nodes outside the group.
  - Importance: Understanding community structure is crucial in various fields, such as social network analysis, biological network analysis, and information retrieval, as it helps to uncover hidden patterns and relationships.

- **Girvan-Newman Algorithm:** *[2]*
  - The Girvan-Newman algorithm aims to find communities in a network by progressively removing edges that are central to the network's connectivity.
  - The idea is that edges connecting different communities will have higher betweenness centrality compared to edges within a community.

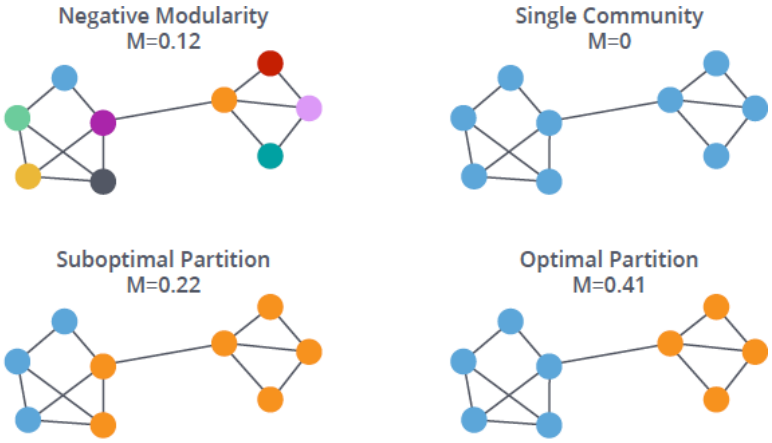
6.2.2 Key Concepts

- **Edge Betweenness Centrality:**



- Definition: Edge betweenness centrality measures the number of shortest paths between pairs of nodes that pass through a given edge. High betweenness centrality indicates that an edge is crucial for connecting different parts of the network.
- Calculation: It involves calculating the shortest paths between all pairs of nodes and counting how many of these paths pass through each edge.

- **Modularity:**



- Definition: Modularity is a measure of the strength of the division of a network into communities. High modularity indicates a network with dense connections within communities and sparse connections between communities.
- Role in Algorithm: The Girvan-Newman algorithm indirectly influences modularity by removing edges that connect different communities, thereby increasing the likelihood of finding well-separated communities.

6.2.3 Step-by-step

1. **Compute Edge Betweenness**

   - ***Shortest Paths Calculation:*** Compute all shortest paths between each pair of nodes in the graph. For unweighted graphs, use Breadth-First Search (BFS). For weighted graphs, use Dijkstra's algorithm.
   - ***Counting Passes:*** Count the number of shortest paths that pass through each edge.
   - ***Betweenness Centrality:*** The betweenness centrality of an edge e is given by:

$$C(e) = \sum_{s \neq t \neq e} \frac{\sigma_{st}(e)}{\sigma_{st}}$$

where σ_{st} is the total number of shortest paths between nodes s and t, and $\sigma_{st}(e)$ is the number of these paths that pass through edge e. This sum is taken over all pairs of nodes s and t that are not endpoints of e.

2. **Remove the Most Important Edge**

   - ***Identify Maximum Betweenness:*** Find the edge with the highest betweenness centrality value.
   - ***Remove Edge:*** Remove this edge from the graph. This edge is considered the most crucial for connecting different parts of the graph.

3. **Recompute Betweenness**

   - ***Update Graph Structure:*** The removal of an edge may affect the betweenness centrality of other edges.
   - ***Recompute Betweenness:*** Recalculate the betweenness centrality for all remaining edges. This involves recomputing shortest paths and counting for the modified graph.

4. **Check for Communities**

   - ***Component Detection:*** Check if the graph has become disconnected into multiple components. This can be done using Depth-First Search (DFS) or BFS to identify connected components.
   - ***Community Identification:*** Each disconnected component represents a community. If the goal is to find a specific number of communities, repeat the process until the desired number of communities is achieved.

5. **Repeat**

   - ***Iterative Removal:*** Recalculate edge betweenness centrality, remove the most important edge, and check for communities iteratively.
   - ***Termination:*** The process can be terminated when the graph is fragmented into the desired number of communities or when further removals do not provide significant insights.

6.2.4 Pseudo Code

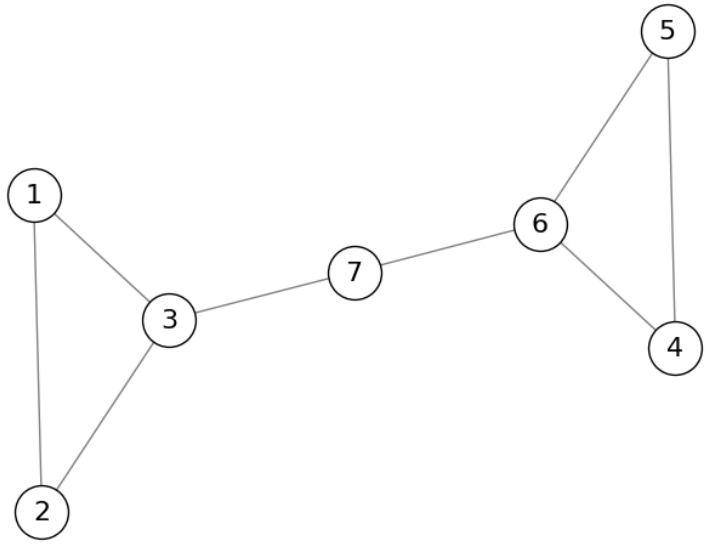
Algorithm 15 Girvan-Newman Algorithm

```
function GirvanNewman(G = (V, E))
    Components ← FindComponents(G)
    while NumberOfComponents(Components) = 1 do
        e ← RemoveEdgeWithHighestBetweenness(G)
        Components ← FindComponents(G)
    end while
    return Components
end function
```

Algorithm 16 Finding Edge Betweenness

```
1: function ComputeEdgeBetweenness(G)
2:     EdgeScores ← Empty map
3:     for each edge e ∈ E do
4:         Score ← 0
5:         for each pair of nodes (s, t) do
6:             Paths ← ShortestPaths(G, s, t)
7:             Count ← CountPathsPassingThrough(e, Paths)
8:             TotalPaths ← NumberOfShortestPaths(s, t)
9:             Score ← Score + Count/TotalPaths
10:        end for
11:        EdgeScores[e] ← Score
12:    end for
13:    return EdgeScores
14: end function
```

6.2.5 Example



Initial Graph:

- **Nodes:** $1, 2, 3, 4, 5, 6, 7$
- **Edges:** $(1, 2), (2, 3), (1, 3), (3, 7), (7, 6), (6, 4), (6, 5), (4, 5)$

1. **Calculate Betweenness Centrality**

Nodes Connected	Shortest Paths Through Edge	Edge
1 – 2	$(1 \to 2)$	(1,2)
1 – 3	$(1 \to 3)$	(1,3)
1 – 4	$(1 \to 3 \to 7 \to 6 \to 4)$	(1,3)(3,7)(7,6)(6,4)
1 – 5	$(1 \to 3 \to 7 \to 6 \to 5)$	(1,3)(3,7)(7,6)(6,5)
1 – 6	$(1 \to 3 \to 7 \to 6)$	(1,3)(3,7)(7,6)
1 – 7	$(1 \to 3 \to 7)$	(1,3)(3,7)
2 – 3	$(2 \to 3)$	(2,3)
2 – 4	$(2 \to 3 \to 7 \to 6 \to 4)$	(2,3)(3,7)(7,6)(6,4)
2 – 5	$(2 \to 3 \to 7 \to 6 \to 5)$	(2,3)(3,7)(7,6)(6,5)
2 – 6	$(2 \to 3 \to 7 \to 6)$	(2,3)(3,7)(7,6)
2 – 7	$(2 \to 3 \to 7)$	(2,3)(3,7)
3 – 4	$(3 \to 7 \to 6 \to 4)$	(3,7)(7,6)(6,4)
3 – 5	$(3 \to 7 \to 6 \to 5)$	(3,7)(7,6)(6,5)
3 – 6	$(3 \to 7 \to 6)$	(3,7)(7,6)
3 – 7	$(3 \to 7)$	(3,7)
4 – 5	$(4 \to 5)$	(4,5)
4 – 6	$(4 \to 6)$	(4,6)
4 – 7	$(4 \to 6 \to 7)$	(4,6)(6,7)
5 – 6	$(5 \to 6)$	(5,6)
5 – 7	$(5 \to 6 \to 7)$	(5,6)(6,7)
6 – 7	$(6 \to 7)$	(6,7)

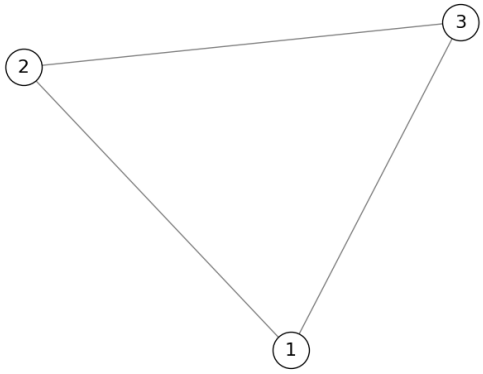
Bảng 1: Shortest Path Through Edges

Edges	Number of path pass through
(1,2)	1
(2,3)	5
(1,3)	5
(3,7)	12
(7,6)	12
(4,5)	1
(6,4)	5
(6,5)	5

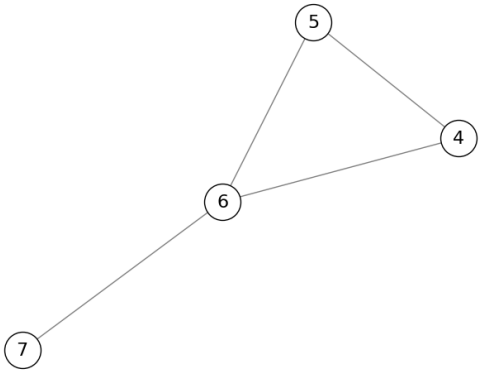
Bảng 2: Edge Betweenness Centrality Calculation

2. **Remove the Edge with Highest Betweenness Centrality**

   Remove the edge $(3, 7)$ or $(7, 6)$, which has the highest betweenness centrality. Assume remove edge $(3, 7)$ The remaining graph include 2 component:



Hình 1: Component 1



Hình 2: Component 2

- **Component 1:**
  - **Nodes:** $1, 2, 3$
  - **Edges:** $(1, 2), (2, 3), (1, 3)$
- **Component 2:**
  - **Nodes:** $4, 5, 6, 7$
  - **Edges:** $(7, 6), (6, 4), (6, 5), (4, 5)$

3. **Recalculate Betweenness Centrality**

   Recalculate the edge betweenness centrality after removing edge $(3, 7)$.

Edges	**Number of path pass through**
(1,2)	1
(2,3)	1
(1,3)	1
(7,6)	3
(4,5)	1
(6,4)	2
(6,5)	2

Bảng 3: Edge Betweenness Centrality Recalculation

4. **Repeat the Process**

   Remove the next edge with the highest betweenness centrality and continue until the graph splits into distinct communities.

6.3 Spectral Clustering

6.3.1 Description

- Spectral Clustering *[7]* is a variant of the clustering algorithm that uses the connectivity between the data points to form the clustering. It uses eigenvalues and eigenvectors of the data matrix to forecast the data into lower dimensions space to cluster the data points. It is based on the idea of a graph representation of data where the data points are represented as nodes and the similarity between the data points are represented by an edge.

- This process mainly involves clustering the reduced data by using any traditional clustering technique – typically K-Means Clustering. First, each node is assigned a row of the normalized of the Graph Laplacian Matrix. Then this data is clustered using any traditional technique. To transform the clustering result, the node identifier is retained.

- This process mainly involves clustering the reduced data by using any traditional clustering technique – typically K-Means Clustering. First, each node is assigned a row of the normalized of the Graph Laplacian Matrix. Then this data is clustered using any traditional technique. To transform the clustering result, the node identifier is retained.

- This clustering technique, unlike other traditional techniques do not assume the data to follow some property. Thus this makes this technique to answer a more-generic class of clustering problems. This algorithm is easier to implement than other clustering algorithms and is also very fast as it mainly consists of mathematical computations.

- The algorithm uses eigenvalue decomposition to reduce the dimensionality of the data, making it easier to visualize and analyze. It is sensitive to noise and outliers, which may affect the quality of the resulting clusters. The algorithm requires the user to specify the number of clusters beforehand, which can be challenging in some cases.

6.3.2 Step-by-step

- Further partitioning can be done using different approaches. For example, we can iterate the clustering procedure over two distinct subgraphs and select a minimum graph slice and then iterate again. Another way is to take the Laplacian eigenvalue mapping and apply K-means. Let's see how this works:

  1. **Compute the normalized Laplacian** based on the formula:

  $$L = D^{-1/2}(D - A)D^{-1/2},$$

  where D is the degree matrix and A is the adjacency matrix.

2. **Stack the eigenvectors of L into a matrix** $(x_1, x_2, \dots)$ in descending order of their corresponding eigenvalues.

3. **Multiply the i-th row by** $\frac{1}{\sqrt{d_i}}$, where d_i is the degree of vertex i.

   – The matrix formed from the k rows (starting from the second row) is called the *Laplacian Eigenmaps* (or *Spectral Embedding*).

4. **Apply K-means clustering** to these eigenmaps.

5. **Assign vertex labels** based on the clusters identified by K-means.

- Pseudo code

Algorithm 17 Spectral Clustering

```
function SpectralClustering(G, n_clusters, n_components)
    A ← Convert graph G to adjacency matrix
    L, degree_sequences ← Compute normalized Laplacian of A
    embedding ← Compute spectral embedding of L using degree_seq and n_components
    kmeans ← KMeans with n_clusters
    Fit kmeans on embedding
    return labels in kmeans
end function
```

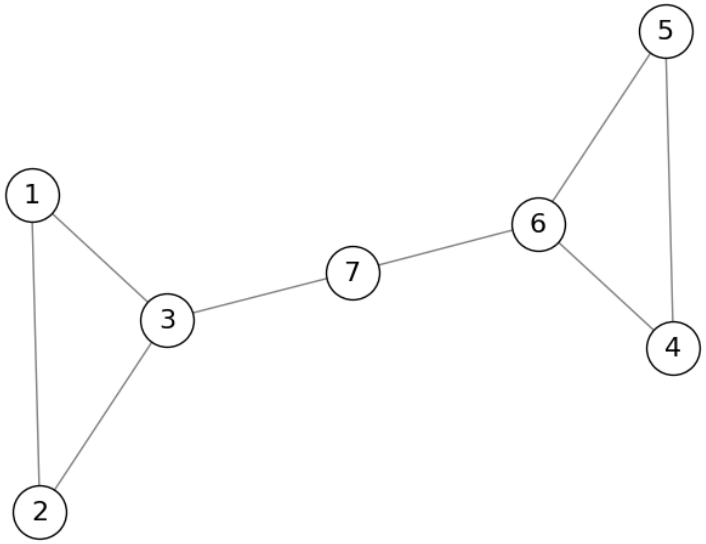
Algorithm 18 Computation of the Normalized Laplacian Matrix

```
function NormLaplacian(A)
    degree_sequence ← Sum of A along the columns
    D ← Diagonal matrix from degree_sequence
    L_norm ← D^{-1/2} × (D − A) × D^{-1/2}
    return L_norm, degree_sequence
end function
```

Algorithm 19 Computation of Spectral Embedding

```
function SpectralEmbedding(L, degree_sequence, n_components)
    X ← Eigenvectors of L × degree_sequence^{-1/2}
    X ← Select columns from column 1^{th} to column(n_components + 1)^{th} of X
    return X
end function
```

6.3.3 Example



1. **Compute the normalized Laplacian** based on the formula:

$$L = D^{-1/2}(D - A)D^{-1/2},$$

where

$$D = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}, A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\Rightarrow \quad L = \begin{pmatrix} 1 & -0.5 & -0.4082 & 0 & 0 & 0 & 0 \\ -0.5 & 1 & -0.4082 & 0 & 0 & 0 & 0 \\ -0.4082 & -0.4082 & 1 & 0 & 0 & 0 & -0.4082 \\ 0 & 0 & 0 & 1 & -0.5 & -0.4082 & 0 \\ 0 & 0 & 0 & -0.5 & 1 & -0.4082 & 0 \\ 0 & 0 & 0 & -0.4082 & -0.4082 & 1 & -0.4082 \\ 0 & 0 & -0.4082 & 0 & 0 & -0.4082 & 1 \end{pmatrix}$$

2. **Stack the eigenvectors of L into a matrix**

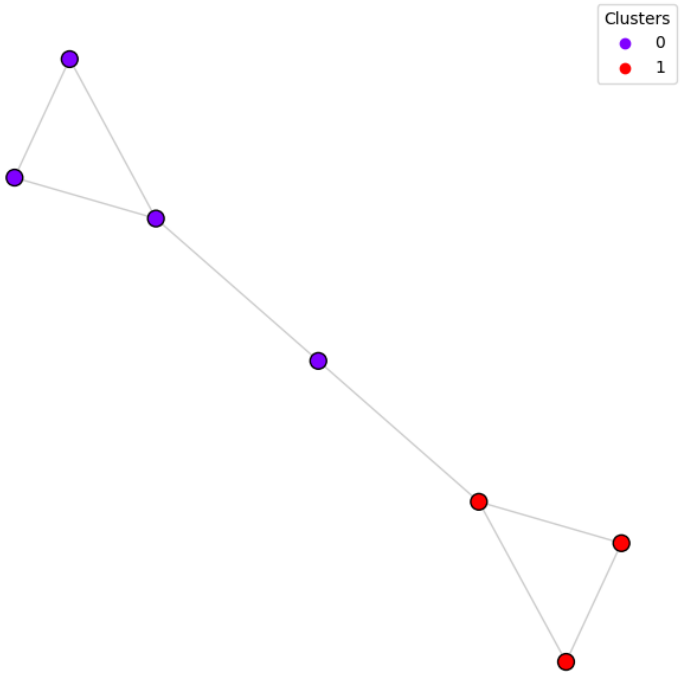
$$Eigenvectors = \begin{pmatrix} -0.3536 & 0.4179 & 0.3084 & -0.2745 & 0.7071 & 0 & 0.1728 \\ -0.3536 & 0.4179 & 0.3084 & -0.2745 & -0.7071 & 0 & 0.1728 \\ -0.433 & 0.3881 & -0.205 & 0.591 & 0 & 0 & -0.5201 \\ -0.3536 & -0.4179 & 0.3084 & 0.2745 & 0 & 0.7071 & 0.1728 \\ -0.3536 & -0.4179 & 0.3084 & 0.2745 & 0 & -0.7071 & 0.1728 \\ -0.433 & -0.3881 & -0.205 & -0.591 & 0 & 0 & -0.5201 \\ -0.3536 & 0 & -0.7317 & 0 & 0 & 0 & 0.5827 \end{pmatrix}$$

3. **Multiply the i-th row by $\frac{1}{\sqrt{d_i}}$**, where d_i is the degree of vertex i. (Assume the number of desired clusters is 2)

$$embedding = \begin{pmatrix} 0.2955 & 0.1781 & -0.1941 \\ 0.2955 & 0.1781 & -0.1941 \\ 0.2745 & -0.1183 & 0.4179 \\ -0.2955 & 0.1781 & 0.1941 \\ -0.2955 & 0.1781 & 0.1941 \\ -0.2745 & -0.1183 & -0.4179 \\ 0 & -0.4225 & 0 \end{pmatrix}$$

4. **Apply K-means clustering** to these eigenmaps. **Assign vertex labels** based on the clusters identified by K-means.

$$labels = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$


Clusters
0
1

References

[1] Réka Albert and Albert-László Barabási. "Statistical mechanics of complex networks". In: *Reviews of Modern Physics* 74.1 (2002). Archived from the original (PDF) on 2015-08-24, pp. 47–97. DOI: 10.1103/RevModPhys.74.47. URL: https://arxiv.org/pdf/cond-mat/0106096.

[2] Michelle Girvan and Mark E. J. Newman. "Community Structure in Social and Biological Networks". In: *Proceedings of the National Academy of Sciences of the United States of America* 99.12 (2002), pp. 7821–7826. DOI: 10.1073/pnas.122653799. URL: https://www.pnas.org/doi/full/10.1073/pnas.122653799.

[3] Ing. Barbora Kabelíková. *Visualization of Knowledge in Education*. Accessed: 2024-08-10. 2023. URL: https://www.fei.vsb.cz/export/sites/fei/470/.content/galerie-souboru/zaverecnePrace/mgr/kabelikova_ing.pdf.

[4] Brian W. Kernighan and Shen Lin. "An efficient heuristic procedure for partitioning graphs". In: *The Bell System Technical Journal* 49.2 (1970), pp. 291–307.

[5] Naveen Ketkar, Lawrence B. Holder, and Diane J. Cook. "Subdue: Compression-Based Frequent Pattern Discovery in Graph Data". In: *Proceedings of the 1st International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations (OSDM '05)*. ACM, 2005, pp. 71–76. URL: https://ailab.wsu.edu/subdue/papers/KetkarOSDM05.pdf.

[6] M. Kuramochi and G. Karypis. "An Efficient Algorithm for Discovering Frequent Subgraphs". In: *IEEE Transactions on Knowledge and Data Engineering* 16.9 (2004), pp. 1038–1051.

[7] Ulrike von Luxburg. *A Tutorial on Spectral Clustering*. Technical Report. https://people.csail.mit.edu/dsontag/courses/ml14/notes/Luxburg07_tutorial_spectral_clustering.pdf. 2007.

[8] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. "DeepWalk: Online Learning of Social Representations". In: *arXiv preprint arXiv:1403.6652v2* (2014). arXiv:1403.6652v2 [cs.SI]. URL: https://arxiv.org/pdf/1403.6652.

[9] Xiaoyang Yan and Jiawei Han. "gSpan: Graph-based Substructure Pattern Mining". In: *Proceedings of the 2002 IEEE International Conference on Data Mining*. IEEE, 2002, pp. 721–724. URL: https://sites.cs.ucsb.edu/~xyan/papers/gSpan-short.pdf.