Section 2: Graph Pattern Matching

UNSW
THE UNIVERSITY OF NEW SOUTH WALES

Faculty of Engineering
Computer Science and Engineering

Outline of Section 2

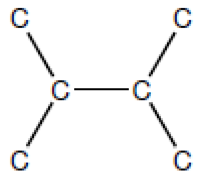
- Introduction
- Given a set of graphs and a pattern graph
  - **G-Index**
  - FG-Index (brief)
  - **QuickSI**
- All-matching
  - **TurboISO**
  - **CFL-Match**
- Distributed Algorithms
- Similarity All-matching

Introduction

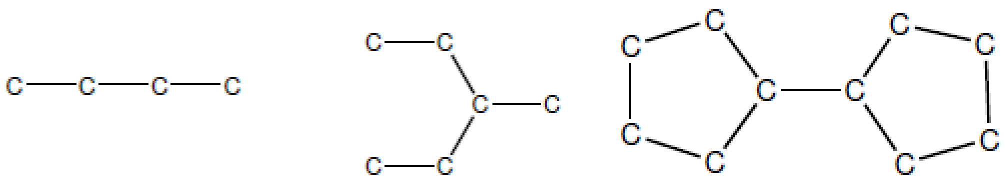
- **Graph Pattern Matching** is an important problem in the graph theory.
- Two categories:

  1. Graph Pattern Matching in Graph Database D

     Given a query pattern, find all graphs in the database D containing this pattern.



Query graph



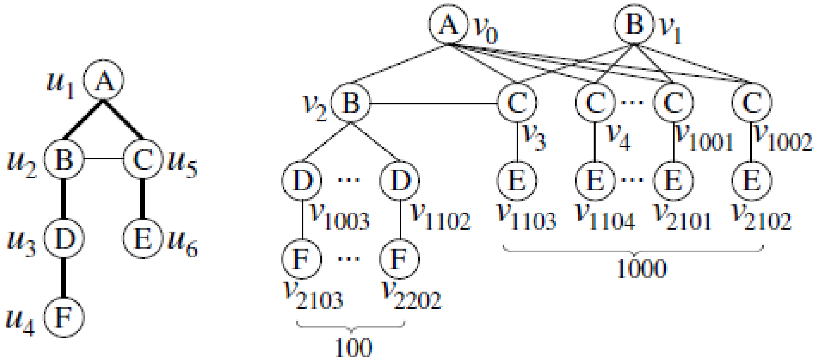
(a) (b) (c)

Graph Database D

Introduction

2. All-Matching

Given a query pattern, enumerate all subgraph embeddings of this pattern in the data graph G.



(a) Query q (b) Data graph G

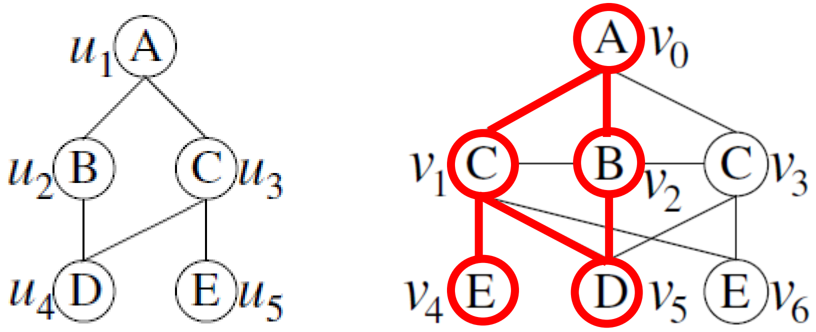
Graph Pattern Matching in Graph Database

- Subgraph Isomorphism

  An subgraph isomorphism is an injective function $f : V(g) \rightarrow V(g')$ such that

$$(1)\ \forall u \in V(g),\ l(u) = l'(f(u))$$
$$(2)\ \forall (u, v) \in E(g)$$

where l and l' are the label function of g and g', respectively.



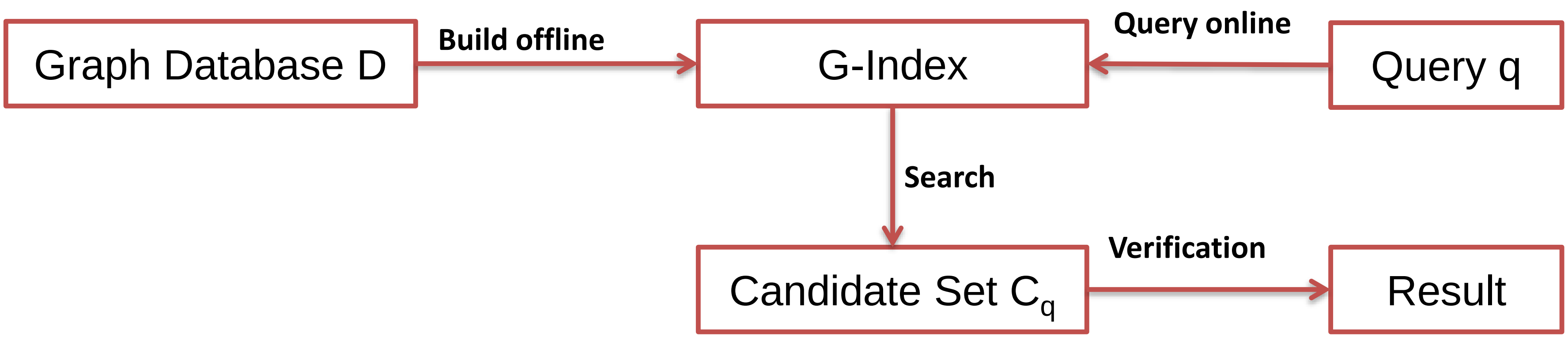
Graph Pattern Matching in Graph Database

- Naive Method
  - Verify all graphs in the graph database D for the given query
  - Infeasible: subgraph isomorphism testing is NP-complete.

- Index-based methods
  - G-Index
  - FG-Index
  - ……

G-Index Framework

- Overview of G-Index framework



- Cost Analysis

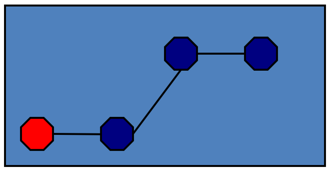
$$\textbf{Query Response Time} = T_{search} + |C_q| * T_{iso\_test}$$

To improve the query response time, we need to minimize:

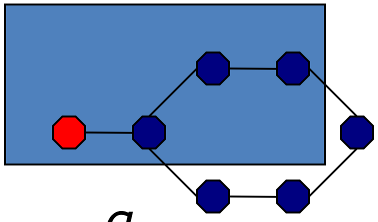
1) the size of graph feature set |F|
2) the size of candidate set C_q

gIndex Pruning

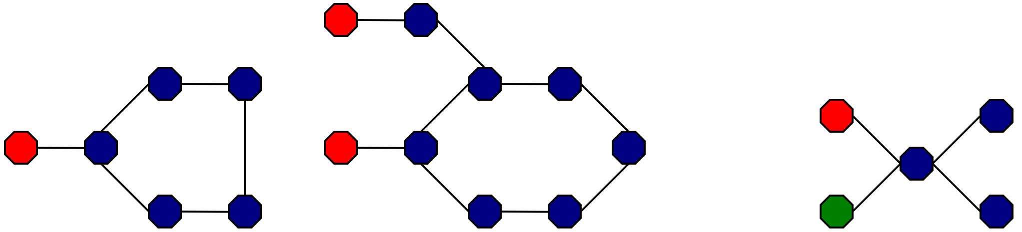
Feature A:



ID-List: $\{g_1, g_2\}$



q



g_1 g_2 g_3

	g_1	g_2	g_3
Filtering:	**Pass** **(Feature A)**	**Pass** **(Feature A)**	**Pruned**
Verification:	**False Positive**	**Answer**	

Substructure Search: gIndex

- Index a set *F* of features from *D*.
- $\forall f \in F$, **D_f :** set of graph ids in *D* contain f

➢**Filtering**: $$C_q = \bigcap_{f \subseteq q \wedge f \in F} D_f$$

➢**Verification**: verify each data graph in C_q.

G-Index Overview

- Index Construction
    - ➢ build an inverted index on the graph feature set F

- Query Processing
    1) Search: query the index to compute the candidate set
    2) Verification: perform subgraph isomorphism test

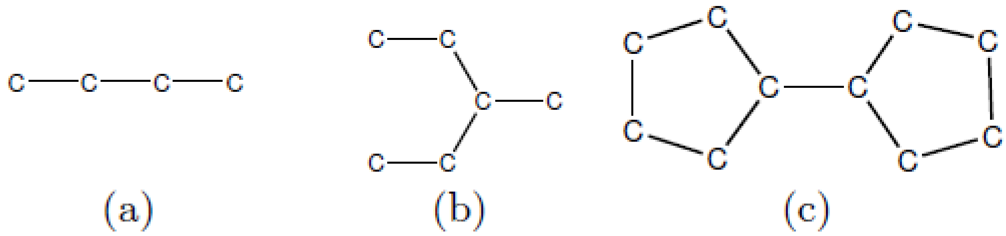
Frequent Fragment

- Frequency

  Given a graph database D, the *frequency* of g, denoted as

  $$freq(g) = |D_g|$$

  , is the number of graphs in D, which contain g as a subgraph.

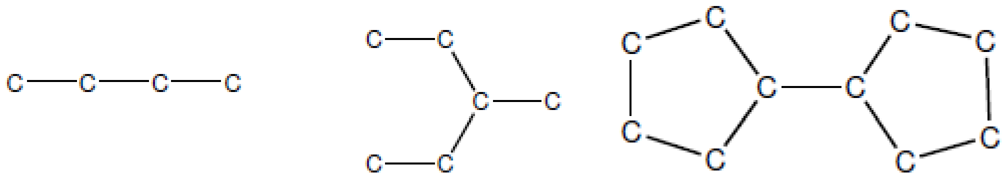


For the graph “c-c”, its frequency in the database consisting of the above three graphs, is 3.

Frequent Fragment

- Frequent graph

A graph/pattern g is *frequent* if its occurrence frequency is no less then a minimum frequency threshold, *minFreq*.



(a) (b) (c)

If *minFreq* is set to 2, then pattern “c-c”, “c-c-c” and “c-c-c-c” are *frequent*.

Can you locate all frequent patterns with *minFreq*=2 in the above given database?

Frequent Fragment

- Uniform *minFreq* is infeasible
  - ➢ In a completely connected graph with 10 vertices, there are 45 1-edge subgraphs, 360 2-edge ones and more than 1,1814,400 8-edge ones.

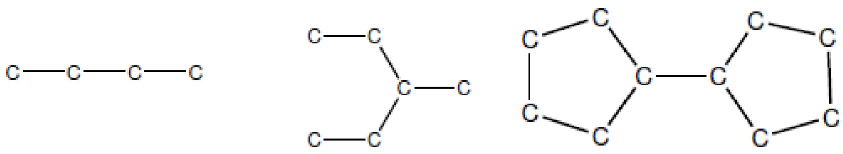
- It's more appropriate to have
  - ➢ Low minimum frequency (threshold) on small fragments (for effectiveness)
  - ➢ High minimum frequency (threshold) on large fragments (for compactness)

- Size-Increasing Frequency
  - ➢ Pattern g is *frequent* if and only if freq(g) ≥ f (size (g)), where freq is the occurrence frequency and f (x) is an increasing function.

Discriminative Fragment

- Do we need to index every frequent fragment?

  All the graphs in the sample database contain carbon-chains: c, c-c, c-c-c, and c-c-c-c. Fragments c-c, c-c-c, and c-c-c-c do not provide more indexing power than fragment c. Thus, they are useless for indexing.



(a) (b) (c)

- Redundant fragment
  - Fragment x is redundant with respect to feature set F if

$$D_x \approx \bigcap_{f \in F \wedge f \subset x} D_f$$

  - c-c, c-c-c, and c-c-c-c are redundant in the above example.

Discriminative Fragment

- Discriminative fragment
  - Fragment x is discriminative with respect to feature set F if

$$D_x << \bigcap_{f \in F \wedge f \subset x} D_f$$

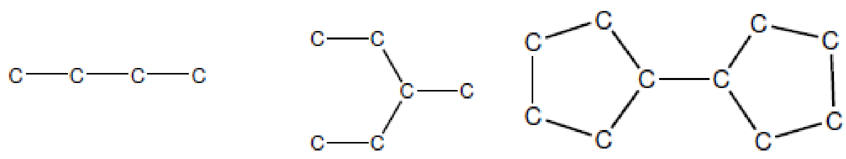
  - In the previous example, graph (a), graph (b) and the carbon ring in graph (c) are discriminative fragments.
- Discriminative ratio

$$\gamma = \frac{|\bigcap_i D_{f_{\varphi_i}}|}{|D_x|}$$

where D_x is the set of graphs containing x and
$\bigcap_i D_{f_{\varphi_i}}$ is the set of graphs which contain the proper subgraphs of x in the feature set

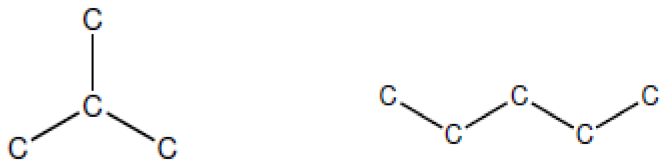
Discriminative Fragment

- Use γ_{min} to mine discriminative fragments
- Example with γ_{min} = 1.5



(a) (b) (c)

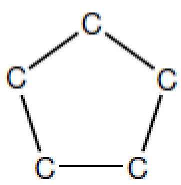
Graph database D



f_1 f_2

Current feature set F

Both f_1 and f_2 contain the subgraphs of this pattern

Given  , its discriminative ratio is $\frac{2}{1} = 2 > \gamma_{min}$

Thus, we add it into the feature set F.

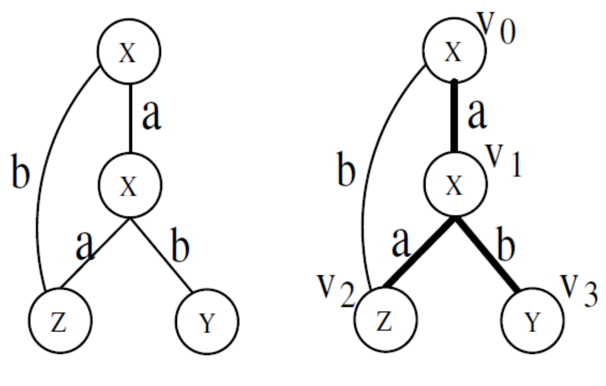
Only graph(c) contains this pattern

G-Index Construction

- Once discriminative fragments are selected, we construct G-Index using the following two steps:
  - Graph Sequentialization
    - Transfer each discriminative fragment into a sequence
  - Build G-Index Tree
    - Store all sequences of discriminative fragments into a prefix tree

Graph Sequentialization

- DFS Code Generation
  - DFS Coding translates a graph into an unique edge sequence, by performing a depth first search (DFS) in a graph.



(a) (b)

Bold edges are the edges of DFS tree originated from the node v_0.

The DFS Code corresponding to the DFS tree in (b) is $\langle (v_0, v_1), (v_1, v_2), (v_2, v_0), (v_1, v_3) \rangle$

Graph Sequentialization

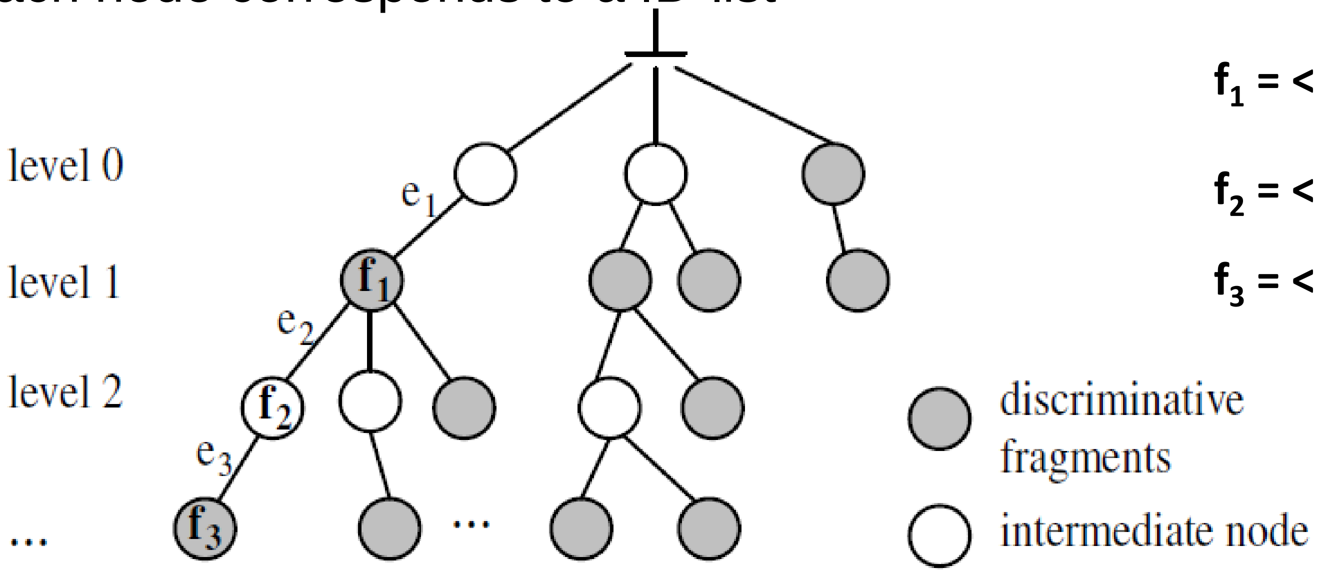
- DFS Code Generation
  - ➢ Represent each edge by a 5-tuple $(i, j, l_i, l_{(i,j)}, l_j)$
    - i and j and id of v_i and v_j
    - l_i and l_j are the labels of v_i and v_j
    - $l_{(i,j)}$ the label of the edge connecting v_i and v_j
  - ➢ The previous DFS code can be represented as

$$\langle\ (0,\ 1,\ X,\ a,\ X)\ (1,\ 2,\ X,\ a,\ Z)\ (2,\ 0,\ Z,\ b,\ X)\ (1,\ 3,\ X,\ b,\ Y)\ \rangle$$

- Canonical label
  - ➢ The minimum DFS code among all of g’s DFS code, denoted by dfs(g) based on the lexicographic order.
  - ➢ If two fragments are the same, they must share the same canonical label.

Build G-Index Tree

- G-Index Tree
  - a prefix tree that store the canonical labels of discriminative fragments
  - Each node corresponds to a ID-list



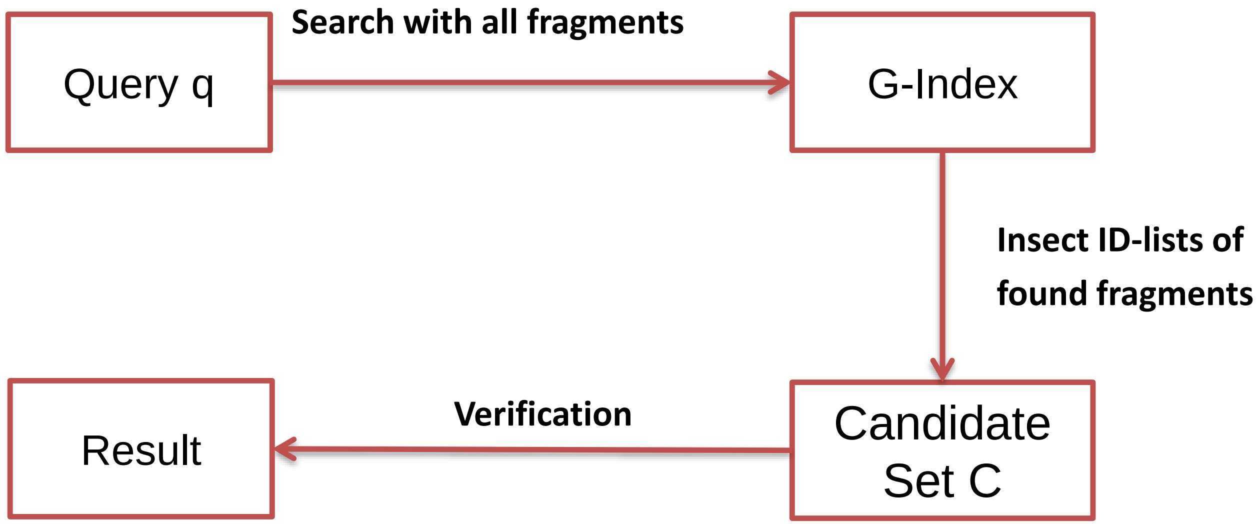
$f_1 = \langle e_1 \rangle$

$f_2 = \langle e_{1,} e_2 \rangle$

$f_3 = \langle e_1, e_2, e_3 \rangle$

  - Some redundant fragments are also stored in the G-Index Tree as intermediate nodes (white nodes).

Query G-Index



1) Enumerate all its fragments of q up to maximum size and locate them in the G-Index.

2) Intersect the ID-lists associated with found fragments to obtain candidate set C.

3) Verify the candidate set C.

Two Rules for Query G-Index

- Apriori Pruning
  - If a fragment is not in the G-Index tree, we need not check its super-graphs any more.

- Maximum Discriminative Fragments
  - If a query q has two fragments, $f_x \subset f_y$, it is not necessary to intersect C_q with D_{f_x}, as

$$C_q \bigcap D_{f_x} \bigcap D_{f_y} = C_q \bigcap D_{f_y}$$

  - Less intersections of ID lists

QuickSI (VLDB2008)

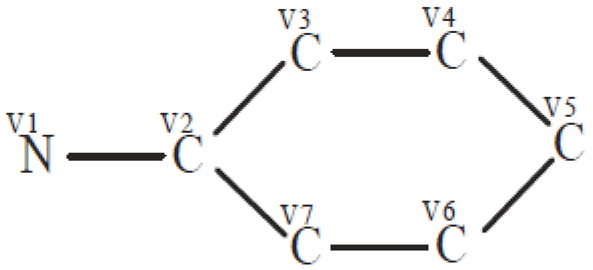
- An Efficient algorithm for Testing Subgraph Isomorphism
  - Proposed for taming verification hardness (NP-complete)
  - A synchronized depth-first traversal technique
  - Three novel pruning techniques

Related Work

- Ullmann Algorithm
  - ➢ First proposed subgraph isomorphism testing.
  - ➢ Random matching order + Backtracking

QI-Sequence

- A sequence that represents a rooted spanning tree for q
  - Basic spanning entries, T_i records basic information
  - Extra entries, R_{ij} records degree/back edges.



q

Type	$[T_i.p, T_i.l]$	$T_i.v$
T_1	$[0,\ N]$	v_1
T_2	$[1,\ C]$	v_2
R_{21}	$[deg : 3]$	
T_3	$[2,\ C]$	v_3
T_4	$[3,\ C]$	v_4
T_5	$[4,\ C]$	v_5
T_6	$[5,\ C]$	v_6
T_7	$[6,\ C]$	v_7
R_{71}	$[edge : 2]$	

Check Degree

Check Back Edge

A possible QI-sequence for q

Three pruning techniques in QI-sequence :

1. **Connected search order**
2. **Degree constraint**
3. **Extra(back) edge constraint**

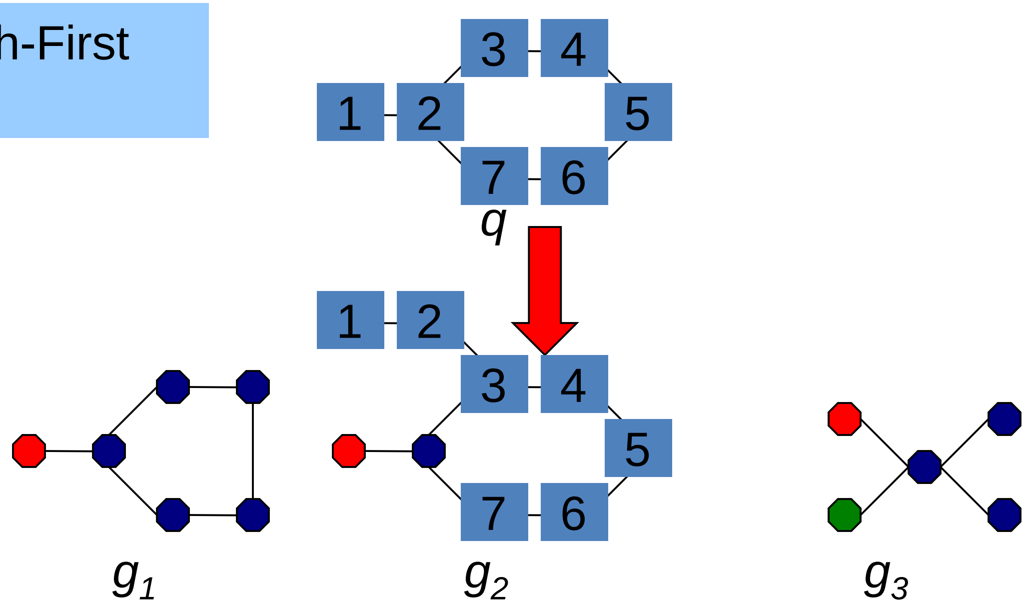
UNSW THE UNIVERSITY OF NEW SOUTH WALES

QuickSI Example

Synchronized Depth-First Traversal

Forwarding

Backtracking



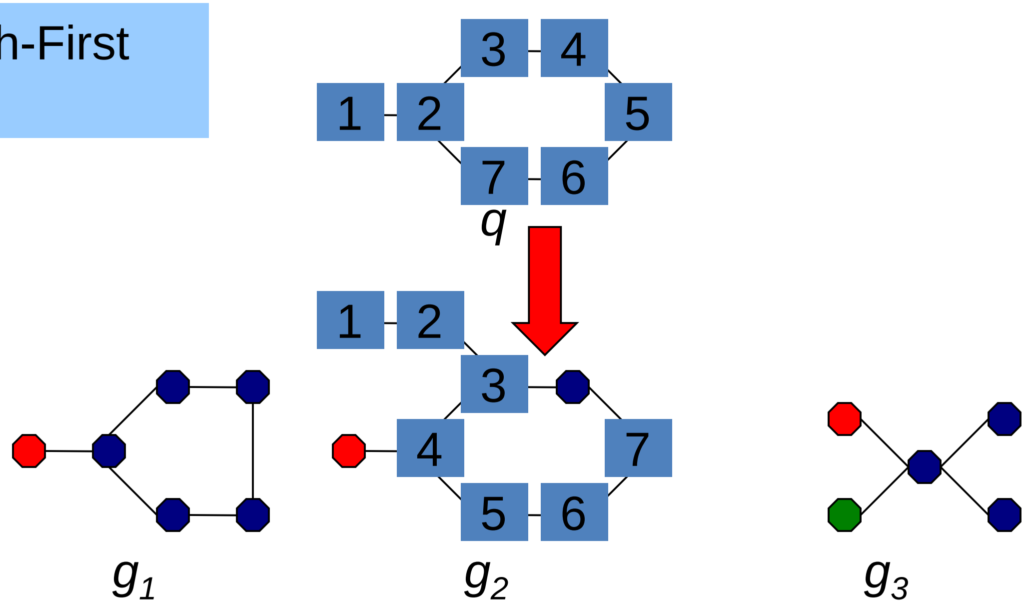
1. Determine the access order in *q*
2. Detect corresponding subgraphs in g_1, g_2 which can be mapped to the currently traversed vertices.

QuickSI Example

Synchronized Depth-First Traversal

Forwarding

Backtracking

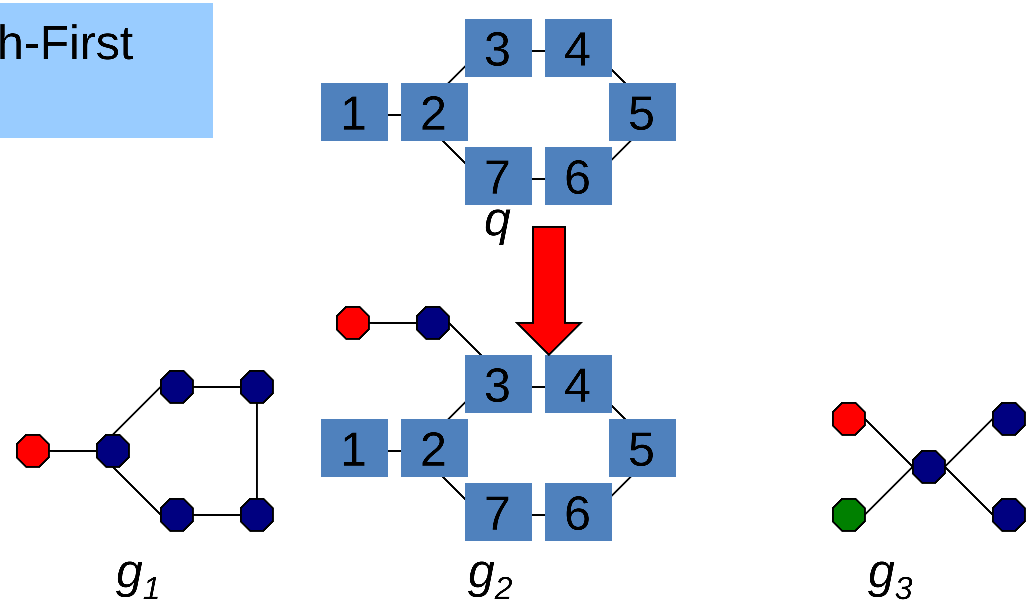


1. Determine the access order of *q*
2. Detect corresponding subgraphs in g_1, g_2 which can be mapped to the currently traversed vertices.

QuickSI Example

Synchronized Depth-First Traversal

Forwarding

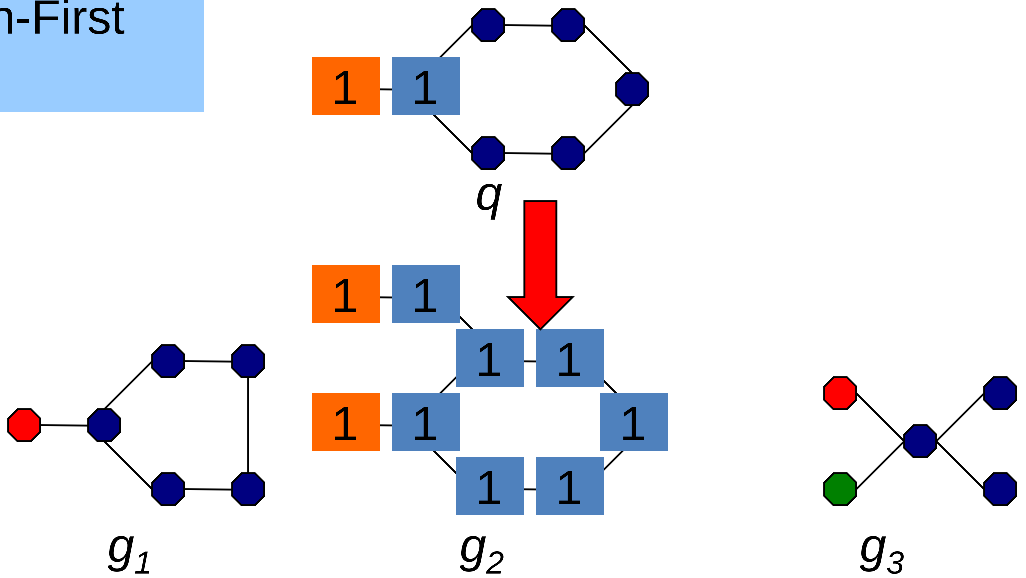


1. Determine the access order of *q*
2. Detect corresponding subgraphs in g_1, g_2 which can be mapped to the currently traversed vertices.

QuickSI Example

Access infrequent labels as early as possible

Synchronized Depth-First Traversal



1. Determine the access order for *q.*
2. Detect corresponding subgraphs in g_1, g_2 which can match the currently traversed vertices.

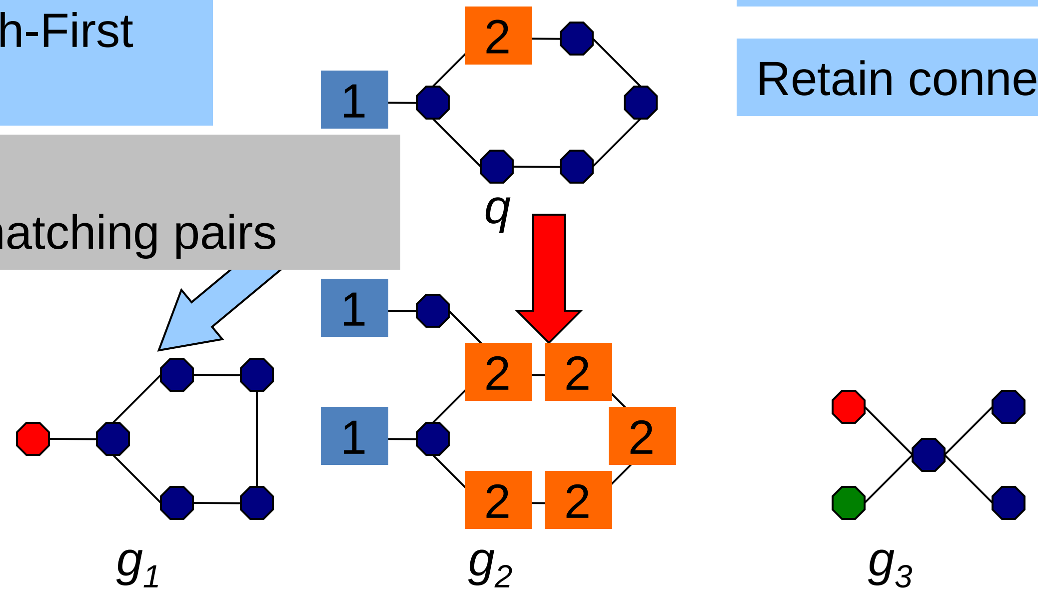
QuickSI Example

Access infrequent labels as early as possible

Retain connectivity

Synchronized Depth-First Traversal

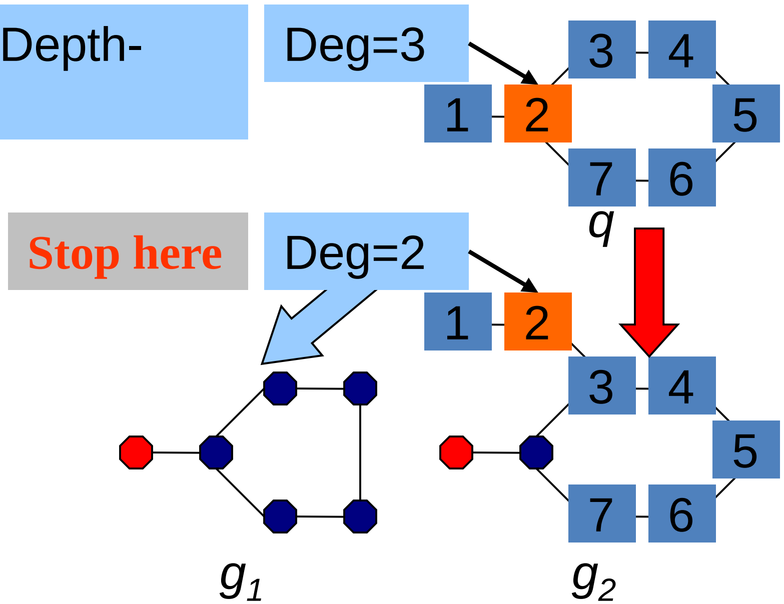
Sparse Graph!
2x5=10 possible matching pairs

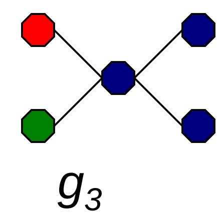


1. Determine the access order for *q*.
2. Detecting corresponding subgraphs in g_1, g_2 which can match the currently traversed vertices.

QuickSI Example

Synchronized Depth-First Traversal





Access infrequent labels as early as possible

Retain connectivity

Effectively use degree information

1. Determine the access order for *q*.
2. Detecting corresponding subgraphs in g_1, g_2 which can match the currently traversed vertices.

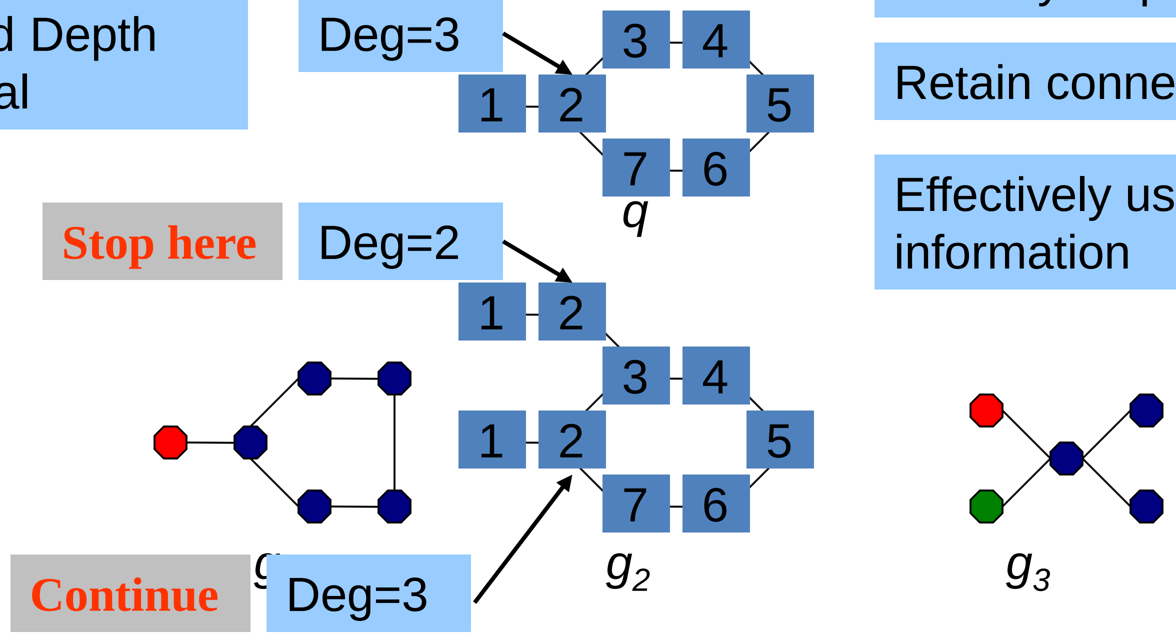
QuickSI Example

Synchronized Depth First Traversal

Access infrequent labels as early as possible

Retain connectivity

Effectively use degree information

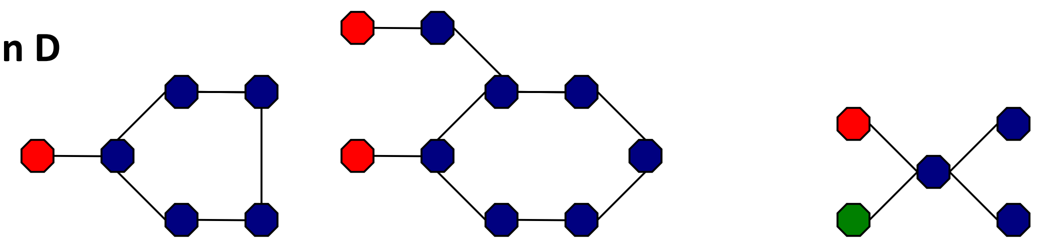


1. Determine the access order for *q.*
2. Detecting corresponding subgraphs in g_1, g_2 which can match the currently traversed vertices.

Swift Index

- A New Filter Approach
  - Precompute **QI-Sequence** for all indexed features(fragments)
    - Only **tree** features are indexed for lower cost of feature mining
  - Index all QI-Sequences in a prefix tree, called **Swift Index**

Given three graphs in D



g_1 g_2 g_3

Mined Features and their ID-List:

ID-List: $\{g_3\}$

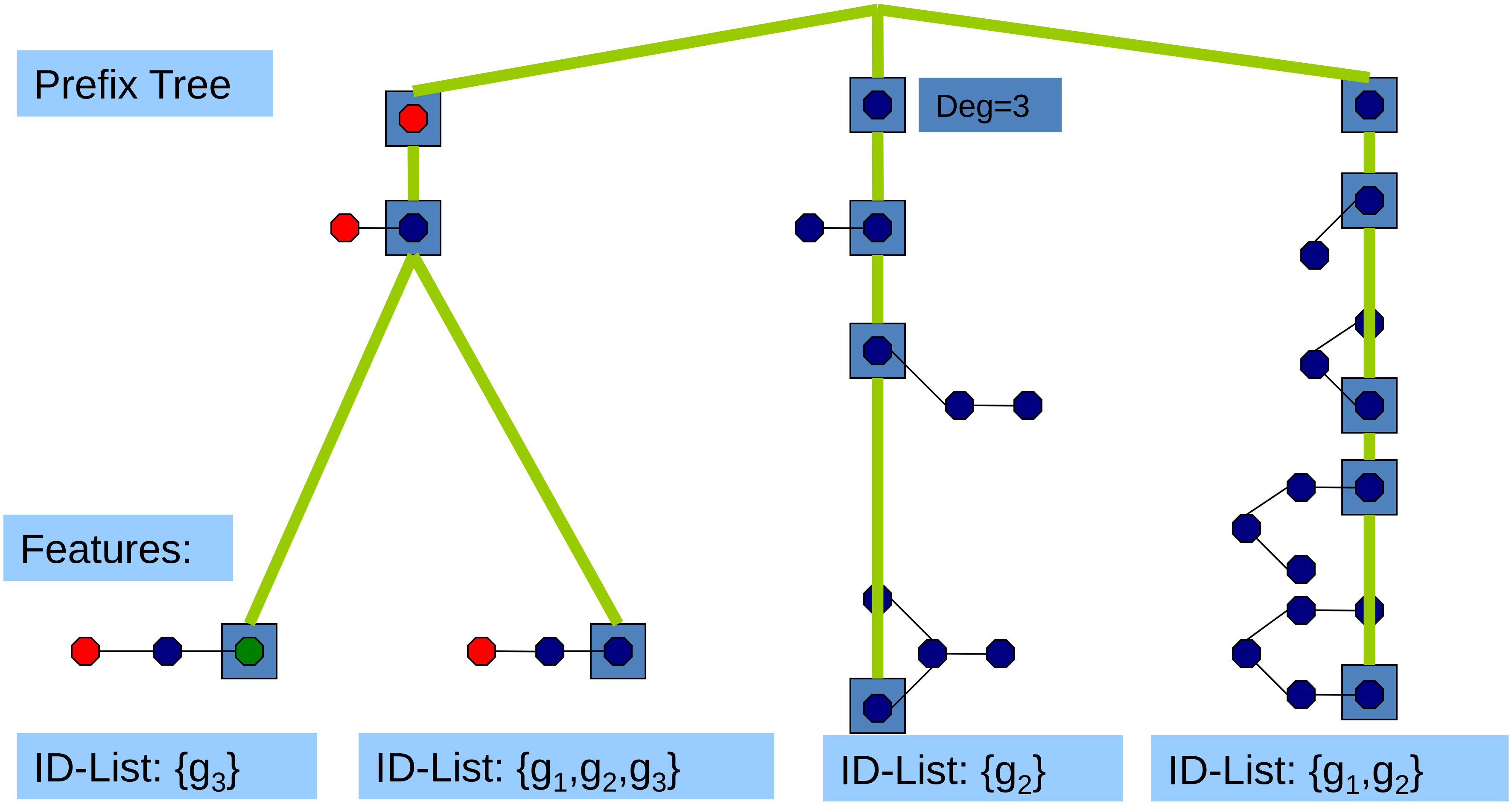
ID-List: $\{g_1,g_2,g_3\}$

ID-List: $\{g_2\}$

ID-List: $\{g_1,g_2\}$

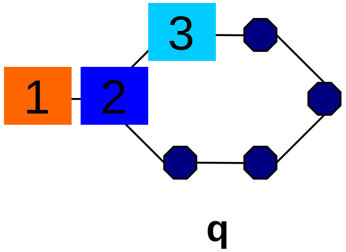
Swift Index

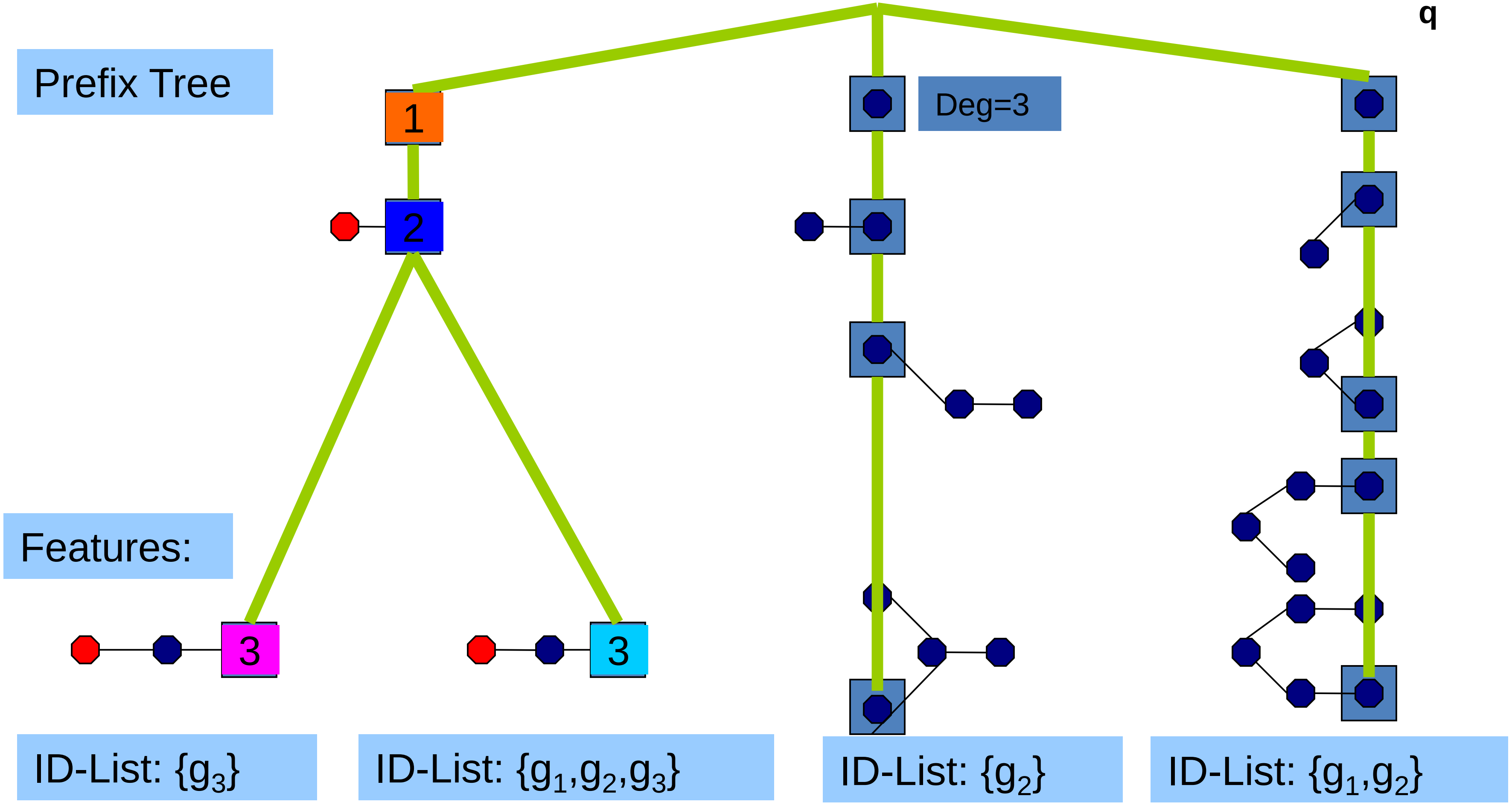
The constructed Swift Index is



- Query Swift Index

Given a query q, traverse the prefix tree from the top to the bottom in the depth-first fashion, to obtain candidate set.





Candidate set for q is $\{g_1,g_2,g_3\}$

Thank you!

Questions?