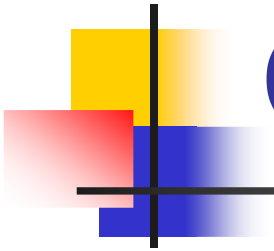
Graph and Web Mining - Motivation, Applications and Algorithms

Prof. Ehud Gudes

Department of Computer Science

Ben-Gurion University, Israel



Course Outline

- Basic concepts of Data Mining and Association rules
  - Apriori algorithm
  - Sequence mining
- Motivation for Graph Mining
- Applications of Graph Mining
- Mining Frequent Subgraphs - Transactions
  - BFS/Apriori Approach (FSG and others)
  - DFS Approach (gSpan and others)
  - Diagonal and Greedy Approaches
  - Constraint-based mining and new algorithms
- Mining Frequent Subgraphs – Single graph
  - The support issue
  - The Path-based algorithm

Course Outline (Cont.)

- Searching Graphs and Related algorithms
  - Sub-graph isomorphism (Sub-sea)
  - Indexing and Searching – graph indexing
  - A new sequence mining algorithm
- Web mining and other applications
  - Document classification
  - Web mining
  - Short student presentation on their projects/papers
- Conclusions

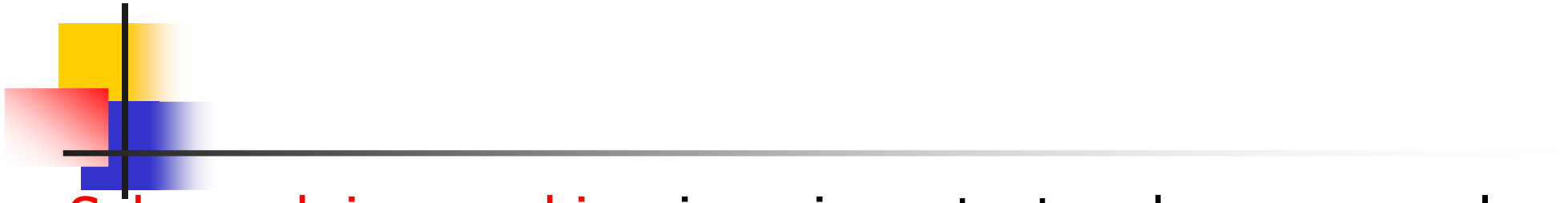
Algorithm for sub-graph isomorphism

Three algorithms will be discussed:

- Ullman
- VF2 – Cordella et. Al.
- Subsea – Lipets, Vanetik, Gudes

The first two will be described very briefly

introduction



- Sub-graph isomorphism is an important and very general form of pattern matching that finds practical application in areas such as:
  - pattern recognition and computer vision,
  - computer-aided design, image processing,
  - graph grammars, graph transformation,
  - Bio-computing,
  - Search operations in chemical structural databases, and numerous others.
  - And of-course: Graph mining
- The subgraph isomorphism problem is generally NP-complete and therefore computationally difficult to solve.

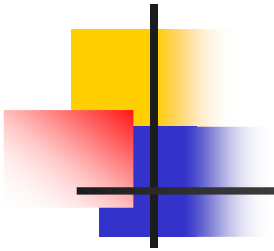
Introduction (Cont.)

Graph mining algorithms often require finding not one but all subgraphs of the database graph isomorphic to a given small graph in order to compute the measure of statistical significance (also called 'support') of that small graph in the database.

The most common technique to establish a subgraph isomorphism is based on backtracking in a search tree. In order to prevent the search tree from growing unnecessarily large, different refinement procedures are used.

Best past known are the algorithm by Ullman and the algorithm by Cordella et al. Cordella is oriented towards finding a single isomorphism. Ullman and Subsea are oriented towards finding all isomorphic occurrences.

Definitions and notations



A graph *G = (V, E) is called vertex-labeled (or simply labeled) if a mapping l : V* →N is given. *l(v) is called a label of a vertex v.*

Two graphs which contain the same number of vertices with the same labels connected in the same way are said to be isomorphic

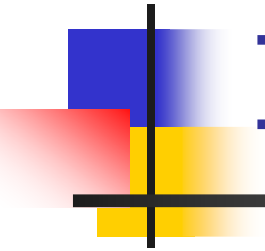
Formally, two graphs *G1 = (V1, E1) and G2 = (V2, E2) are isomorphic, denoted by G1 =~ G2, if there is a (label-preserving) bijection ϕ : V1 −→ V2 such that, for every* pair of vertices *vi, vj ∈ V1, (vi, vj) ∈ E1 if and only if ϕ(vi), ϕ(vj) ∈ E2. Bijection ϕ is said to be an isomorphism between two graphs.*

A graph *G′ is a subgraph of a given graph G if vertices and edges of G′ form* subsets of the vertices and edges of *G.*

A graph *G1 = (V1, E1) is isomorphic to a subgraph of a graph G2 = (V2, E2) if* there exists a subgraph of *G2, say G2a*, such that *G1 =~ G2a*

Subgraph isomorphism – a Naïve Algorithm

- A graph *G1 = (V1, E1) is isomorphic to a subgraph of a graph G2 = (V2, E2) if* there exists a subgraph of *G2, say G2a*, such that *G1 =~ G2a*
- *How can we find G2a?*
- *Assume G1 has n nodes. Lets examine each subset of G2 that has n nodes, check if they have the same labels as nodes in G1, and if yes, check if the edge in G1 exists also in the selected set.*
- *Obviously an exponential algorithm!*



An Algorithm for Subgraph Isomorphism

J. R. ULLMANN, 1976

The enumeration algorithm

- To find isomorphism we need to find a correspondence between vertices such that the adjacency matrix will be identical.
- Assume A and B are the adjacency matrices of G and G' respectively. The problem is to find a subgraph in G' isomorphic to G
- A matrix *M'* *(whose elements are 0 and 1) can be used to*

  permute the rows and columns of B to produce a further matrix C. Specifically, we define

  C = *M'(M'B)^T, where T denotes transposition. If it is true that*

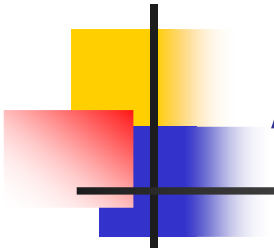
  (ViVj) ($a_{i,j}$= 1) => ($c_{i,j}$ = 1) and the labels are equal

  Then M' specifies an isomorphism between G, and a subgraph of G'.
- The main problem is enumerating all the possible M' matrices

Algorithm Employing Refinement Procedure

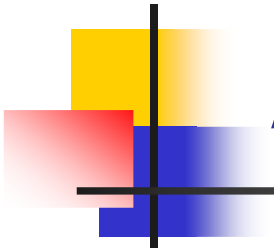
- We start with a matrix with many 1's meaning that any node can map to any node.
- To reduce the amount of computation required for finding subgraph isomorphism we employ a procedure, which we call the ***refinement procedure****, that eliminates some of the* 1's from the matrices *M, thus eliminating successor nodes in the search tree.*
- Ullmann's algorithm attains efficiency by eliminating successor nodes in the search tree.
- the original part of the algorithm consists of a procedure that is entered after each node in the search tree. The result of this procedure is generally a reduction in the number of successor nodes that must be searched, which yields a reduction in the total computer time required for determining isomorphism

Algorithm Employing Refinement Procedure – cont(1)



- We say that an isomorphism is an **isomorphism *under* M** if its terminal node in the search tree is a successor of the node with which M is associated.
- The 0's in the matrix M merely preclude correspondences between nodes.
- Our goal is to preclude as many nodes as possible, which means that we like to be able to change $m_{ij} = 1$ to $m_{ij} = 0$ without losing any of the isomorphism's under M: all such isomorphism's will still be found by the tree search.

Algorithm Employing Refinement Procedure – cont(2)

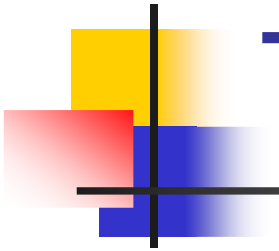


- Generally the result of the refinement procedure is to change some of the I's in M to O's. This corresponds to a non-match because of no corresponding edge.
- The check whether a 1 is changed to zero is made by considering all the adjacent nodes to the current node. If they are not also 1, then the original '1' is wrong
- During the refinement procedure we continually check whether any row of M contains no 1.
- If any row of M contains no 1 then the procedure jumps to its FAIL exit, because there is no advantage in continuing the procedure. Otherwise the procedure terminates at its SUCCEED exit.

VF2 - A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs

Luigi P. Cordella, Pasquale Foggia, Carlo Sansone,

and Mario Vento, 2004



THE VF2 ALGORITHM

Assume the problem is to find a subgraph in G1 isomorphic to the graph G2.

The main idea is to construct a state S which contains a correct partial match between nodes of G1 and G2

M(s) identifies two sub graphs of G1 and G2, say G1(s) and G2(s), obtained by selecting from G1 and G2 only the nodes included in M(s), and the branches connecting them. Where s is a state of the matching process.

The main problem is extending M(s) with new branches.

An extension of S is adding a pair (n,m) where n belongs to G1 and m belongs to G2.

Feasibility rules is a set of rules that are able to verify the consistency conditions, making possible the generation of consistent states only.

THE VF2 ALGORITHM (con.)

- if **F(s,n,m)** is consistent, being p=(n,m), the successor state s' =s U p is computed and the whole process recursively applies to s'.

- That is for each possible successor state the feasibility rules are checked and if found consistent the state is extended

- The set **P(s)** of all the possible pairs candidate to be added to the current state is obtained by considering first the sets of the nodes directly connected to G1(s) and G2 (s).

The match procedure

```
PROCEDURE Match(s)
    INPUT:  an intermediate state s; the initial state s0 has M(s0)=∅
    OUTPUT: the mappings between the two graphs

    IF M(s) covers all the nodes of G2 THEN
      OUTPUT M(s)
    ELSE
      Compute the set P(s) of the pairs candidate for inclusion in M(s)
      FOREACH p in P(s)
        IF the feasibility rules succeed for the inclusion of p in M(s) THEN
          Compute the state s´ obtained by adding p to M(s)
          CALL Match(s')
        END IF
      END FOREACH
      Restore data structures
    END IF
END PROCEDURE Match
```

THE VF2 ALGORITHM (con 3)

- Five feasibility rules are defined: Rpred, Rsucc, Rin, Rout, and Rnew.
- The first two rules check the consistency of the partial solution M(s') obtained by adding the considered candidate pair (n,m) to the current partial solution M(s).
- The remaining three rules are introduced for pruning the search tree; in particular, Rin and Rout perform a 1-look-ahead in the searching process, and Rnew a 2-lookahead.
- For example, the first rule checks whether for each predecessor of n in G1 there is such predecessor of m in G2, and vice-versa.

$$F_{\text{syn}}(s, n, m) = R_{\text{pred}} \wedge R_{\text{succ}} \wedge R_{\text{in}} \wedge R_{\text{out}} \wedge R_{\text{new}}.$$

The Rules

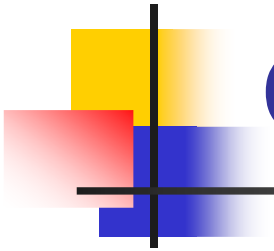
$$R_{\text{pred}}(s,n,m) \Longleftrightarrow$$
$$(\forall n' \in M_1(s) \cap \text{Pred}(G_1,n) \exists m' \in \text{Pred}(G_2,m) \mid (n',m') \in M(s)) \wedge$$
$$(\forall m' \in M_2(s) \cap \text{Pred}(G_2,m) \exists n' \in \text{Pred}(G_1,n) \mid (n',m') \in M(s)),$$

$$R_{\text{succ}}(s,n,m) \Longleftrightarrow$$
$$(\forall n' \in M_1(s) \cap \text{Succ}(G_1,n) \exists m' \in \text{Succ}(G_2,m) \mid (n',m') \in M(s)) \wedge$$
$$(\forall m' \in M_2(s) \cap \text{Succ}(G_2,m) \exists n' \in \text{Succ}(G_1,n) \mid (n',m') \in M(s)),$$

$$R_{\text{in}}(s,n,m) \Longleftrightarrow$$
$$(\text{Card}(\text{Succ}(G_1,n) \cap T_1^{\text{in}}(s)) \geq \text{Card}(\text{Succ}(G_2,m) \cap T_2^{\text{in}}(s))) \wedge$$
$$(\text{Card}(\text{Pred}(G_1,n) \cap T_1^{\text{in}}(s)) \geq \text{Card}(\text{Pred}(G_2,m) \cap T_2^{\text{in}}(s))),$$

$$R_{\text{out}}(s,n,m) \Longleftrightarrow$$
$$(\text{Card}(\text{Succ}(G_1,n) \cap T_1^{\text{out}}(s)) \geq \text{Card}(\text{Succ}(G_2,m) \cap T_2^{\text{out}}(s))) \wedge$$
$$(\text{Card}(\text{Pred}(G_1,n) \cap T_1^{\text{out}}(s)) \geq \text{Card}(\text{Pred}(G_2,m) \cap T_2^{\text{out}}(s))),$$

$$R_{\text{new}}(s,n,m) \Longleftrightarrow$$
$$\text{Card}(\tilde{N}_1(s) \cap \text{Pred}(G_1,n)) \geq \text{Card}(\tilde{N}_2(s) \cap \text{Pred}(G_2,n)) \wedge$$
$$\text{Card}(\tilde{N}_1(s) \cap \text{Succ}(G_1,n)) \geq \text{Card}(\tilde{N}_2(s) \cap \text{Succ}(G_2,n)).$$



Cordella – Experimental results

- Cordella compared their algorithm to two algorithms: Ullman and Nauty, where Nauty is an algorithm that uses some form of cannonical labeling

- There was not a clear winner for all tested graphs

- Citation: From the analysis of the table, it appears that Nauty is more convenient on randomly connected graphs that exhibit no regular structure, especially when the edge density becomes high. This kind of graph, anyway, does not adequately represent the graph structures found in many applications, where the graphs often show some form of regularity. On the other hand, graphs with a more regular structure, VF2 is more efficient, especially for large graph sizes

Subsea

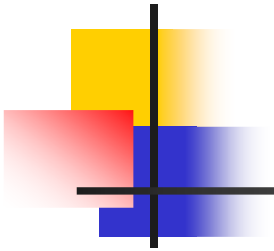
An efficient heuristic algorithm for Subgraph isomorphism –
Lipets, Vanetik, Gudes, 2008

Definition and notations

A graph G = (V,E) is called ***vertex-labeled*** *if a mapping* $l : V \rightarrow L$ is given.

Two graphs which contain the same number of vertices with the same labels connected in the same way are said to be **isomorphic**.

Definitions (Cont.)



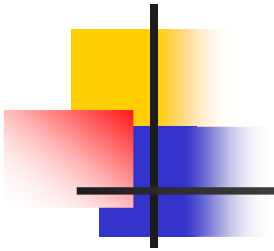
- An ***induced subgraph*** *is a subset of the vertices of a graph G together with* **all** edges whose endpoints are both in this subset. Formally, let G be a graph and $V' \subset V(G)$. We call the graph $G' = (V', E(G) \cap \{(u, v) | u, v \in V'\})$ the *subgraph of G induced by V ' and we denote it by G(V '). The relationship* between G′ and G in this case we denote by $\sqsubseteq$.

- an **induced subgraph isomorphism** is an isomorphism with an induced subgraph of a given graph, i.e.,

  a graph G1 = (V1,E1) is isomorphic to an induced subgraph of a graph G2 = (V2,E2) if there exists an induced subgraph of G2, say G′2, such that G1 ~=G′2.

Subsea deals with both kinds of isomorphism

Definitions (Cont.)



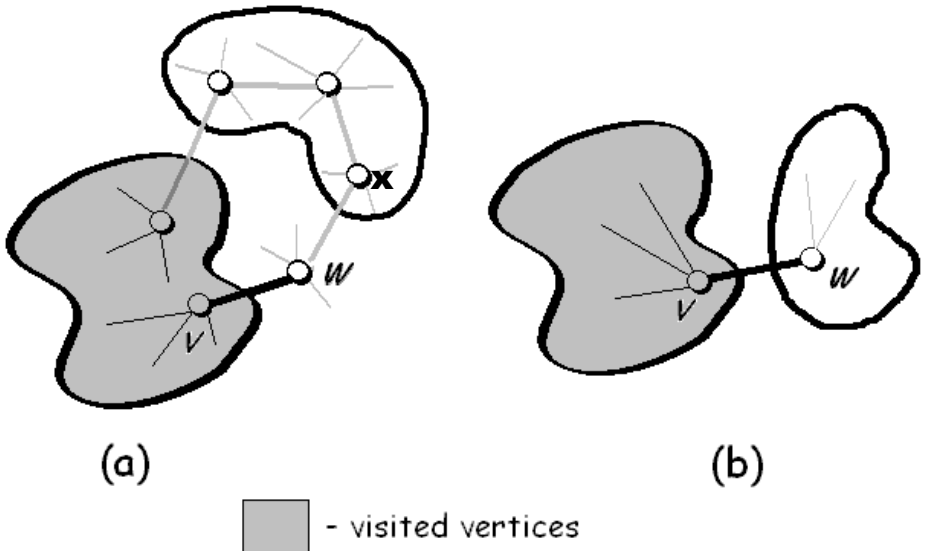
- The **neighborhood** of a vertex v in graph G, denoted by NG(v), is the set of vertices in G that are adjacent to v, i.e., NG(v) = {u ∈ V |(u, v) ∈ E}. For any e ∈ E(G), we define G − e = (V (G),E(G) \ {e}).

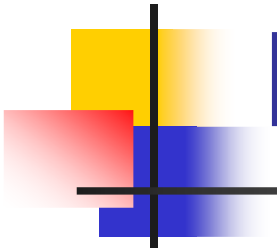
- The **size of the cut** $(A,\bar{A})$, denoted by $c(A,\bar{A})$, is the number of edges having exactly one vertex in A and the other in $\bar{A}$, namely $|e(A,\bar{A})|$.

- The **minimum bisection** is a cut $(A,\bar{A})$ minimizing $c(A,\bar{A})$ over all sets with A of size $\lceil |V|/2 \rceil$. For arbitrary graphs G, the problem of determining the minimum bisection is NP-hard.

Outline of the Subsea algorithm

1. In a preprocessing step generate all the traverse histories of the pattern graph. (will be explained later)
2. Decompose the target graph by finding an approximate minimum bisection (heuristically).
3. Check all possible isomorphisms using edges belonging to the bisection. (e.g. (v,w) and (w,x))
4. Apply step 2 again recursively on the two parts of the bisected graph until the target graph becomes comparable or smaller in size to the “small” graph.





Bisection algorithms

- two well-known approximation algorithms for finding a minimum bisection of a given graph G:
    1. Black Holes Bisection algorithm.
    2. Simple Greedy Bisection method.
- Note that since the minimum bisection is only a tool to decompose the problem efficiently, we are not required to find the actual minimum bisection (which is a hard problem), but it is enough to provide an approximation for it.

Black Holes Bisection algorithm

- Given a graph G = (V,E), the algorithm runs as follows. Initialize B1 = B2 = ∅. These are the black holes.
- Choose uniformly at random an edge from V \ (B1 ∪ B2) to B1, and add the new endpoint to B1. If no such edge exists, choose uniformly at random among all vertices in V \ (B1 ∪ B2) for a vertex to add to B1. Do the same for B2. Repeat until |B1 ∪ B2| = |V |.

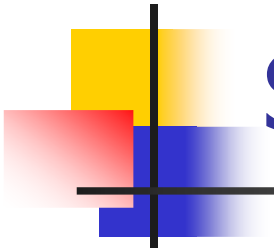
Black Holes bisection (Cont.)

Black Holes Bisection
Input: Graph G = (V,E)
Output: Cut (B, $\bar{B}$) of V which approximates a minimal bisection

1: $B1 \leftarrow B2 \leftarrow \varnothing$
2: $B0 \leftarrow V \setminus (B1 \cup B2)$ /* initially B0 is the whole set of nodes */
3: repeat
4: Add2Hole(1)
5: Add2Hole(2)
6: until $B0 = \varnothing$

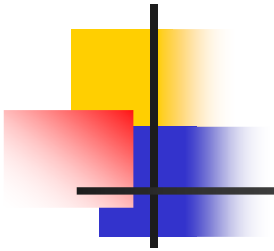
Procedure Add2Hole(i):

1: if $B0 = \varnothing$ return
2: $E0 \leftarrow \{(u, v) : u \in Bi, v \in B0\}$
3: if $E0 \neq \varnothing$ then
4: chose randomly $e = (u, v) \in E0$ with $v \in B0$
5: else
6: chose randomly $v \in B0$
7: end if
8: $Bi \leftarrow Bi \cup \{v\}$
9: $B0 \leftarrow B0 \setminus \{v\}$



Simple Greedy Bisection method

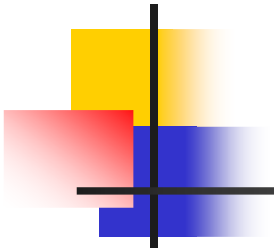
- The obvious greedy algorithm for the graph bisection problem consists of starting with any bisection $(B, \bar{B})$ of V and computing a new bisection by swapping the pair of elements $x \in B$, $y \in \bar{B}$ which maximizes the gain (number of edges in $(B, \bar{B})$ before the swap minusthe number after the swap – if this number is positive then we reduced the size of the cut…).
- This process is repeated until the maximum gain is less than zero or until the maximum gain is zero and another heuristic has determined that it is time to stop swapping "zero gain" pairs.
- Assuming we have a good bisection lets look at how we search for isomorphism



Traverse history

- We will see two heuristic methods to represent a "small" pattern graph.
- Each such representation enumerates the vertices of the pattern graph in a particular order. This order will determine the order in which the isomorphism check is done.
- Somewhat similar to "canonical labeling" but not so complex…

Traverse history (Cont.)



- We will see two heuristic methods to represent a "small" pattern graph.

- Each such representation enumerates the vertices of the pattern graph in a particular order. This order will determine the order in which the isomorphism check is done.

Traverse history (Cont.)

Let $d : V \longrightarrow N$ be a numbering of vertices of graph G.

Let li denote the label of the vertex that has number i in numbering d, i.e.,

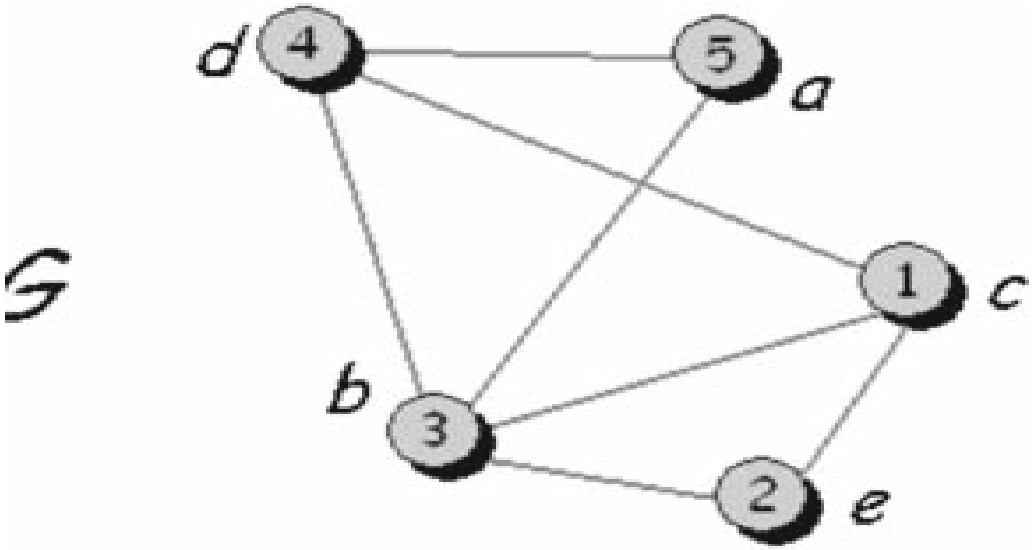
li := l(v), d(v) = i; let Ni := {d(u) < i : u $\in$ NG(v), d(v) = i}.

The sequence $(l_1,N_1), (l_2,N_2), \ldots, (l_{|V|},N_{|V|})$ is called a ***traverse history of graph*** **G** induced by numbering d.

Informally, Ni is the set of adjacent vertices to i with numbering smaller than i.

Our goal is to traverse the graph in such an order that we always prefer nodes that have high connectivity to the already selected nodes

Traverse history (Cont.)



$\langle \underbrace{\{\}}_{N_1}, \underbrace{\{1\}}_{N_2}, \underbrace{\{1,2\}}_{N_3}, \underbrace{\{1,3\}}_{N_4}, \underbrace{\{1,3,4\}}_{N_5} \rangle$

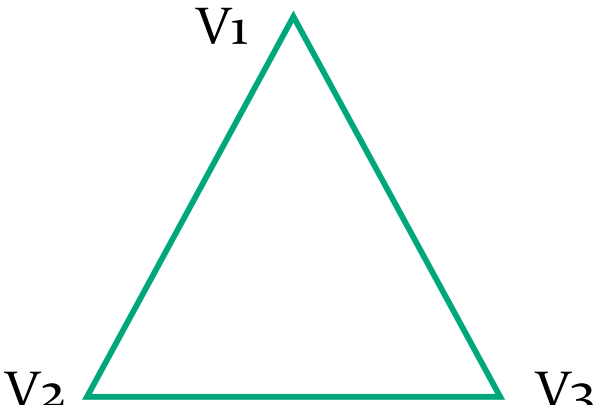
{3,4} **will come next because 3 has a high degree**

Traverse History - The DFS approach

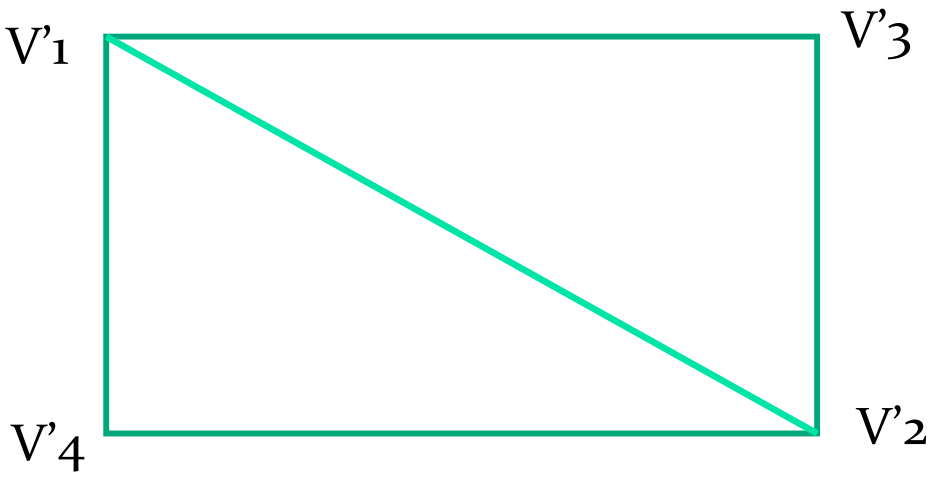
Traverse History
Input: Graph G = (V,E), starting vertices v1, v2 ∈ V , with (v1, v2) ∈ E
Output: Traverse history H started on v1, v2 ∈ V .
1: for all v ∈ V do
2: d(v) ←− 0
3: end for
4: vtime ←− etime ←− 1
5: V isit(v1)
6: return H
Procedure V isit(v)
1: d(v) ←− vtime
2: H[vtime + +] = (l(v), {0 < d(u1) ≤ ... ≤ d(um) : u1, ..., um ∈ NG(v)})
3: if v = v1 then V isit(v2)
4: N0 ←− {u ∈ NG(v) : d(u) = 0}
5: while N0 # ∅ do
6: choose w ∈ N0 with lexicographically minimal
EstimateNext(w, v) /* choose the node with high proximity */
pair
7: if d(w) = 0 then V isit(w)
8: N0 ←− N0 \ {w}
9: end while

Search technique

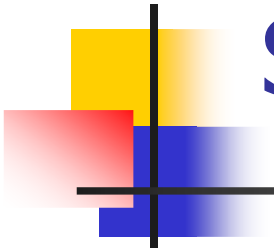
- The algorithm receives as an input "large" target graph GL = (VL,EL), starting vertices v1, v2 ∈ VL and the traverse history H of a "small" pattern graph GS.
- It finds all subgraphs (of GL (v1 →v'1 , v2 → v'2)-isomorphic to GS, where v'1 , v'2 are the first two vertices of the traverse history H.
- Note that by scanning high degree nodes first, we will fail to find edges often and exit the search early….



TH = <0 , {1}, {1,2}>



TH = < 0, {1}, {1,2},{1,2}>



Search Technique – main theorem

- When a traverse history is found in the tested graph which is equal to the traverse history of the pattern graph – that means an isomorphism was found
- A good traverse history will cause the search procedure to fail early which will minimize the search time – see details in paper

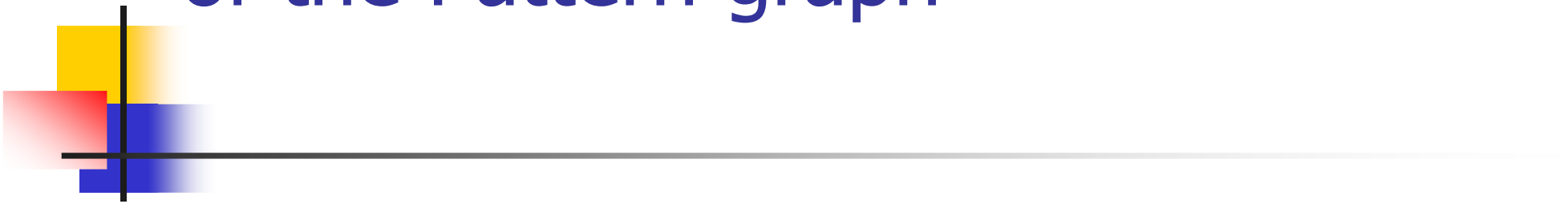
Subsea: Subgraph Isomorphism Algorithm

- Precomputation stage:

  A pair of vertices (v1, v2) $\in$ V 2 of graph G we will call redundant if there exists an (v1 $\rightarrow$ v′1, v2 $\rightarrow$ v′2)-automorphism of G. We look only for traverse histories that start with non-redundant nodes.

- Algorithm 6.1 finds a corresponding traverse history for each non-redundant pair of adjacent vertices. Note that each edge of the pattern graph may derive 0, 1, or 2 traverse histories (depending on the numbering)
- So for each pattern graph we derive several traverse histories, each starts with a different non-redundant edge. We store all these traverse histories in a ready data structure.

Generating all traverse histories of the Pattern graph



Alg. 6.1: All Traverse Histories

Input: Graph G = (V,E)

Output: Set of traverse histories of G

1: A $\leftarrow -$ $\varnothing$

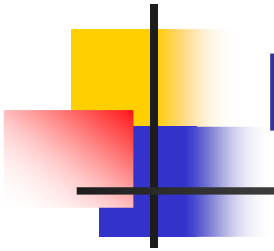
2: for each (v1, v2) $\in$ V 2 such that (v1, v2) $\in$ E do

3: run Algorithm 4.1 on G, v1, v2 to obtain traverse history $H_{v1,v2}$

4: if ! IsRedundant(v1, v2) then A $\leftarrow -$ A $\cup$ $\{H_{v1,v2}\}$

5: end for

6: return A



Finally - Main algorithm

Find the traverse history for each non-redundant pair of adjacent vertices of the pattern graph.

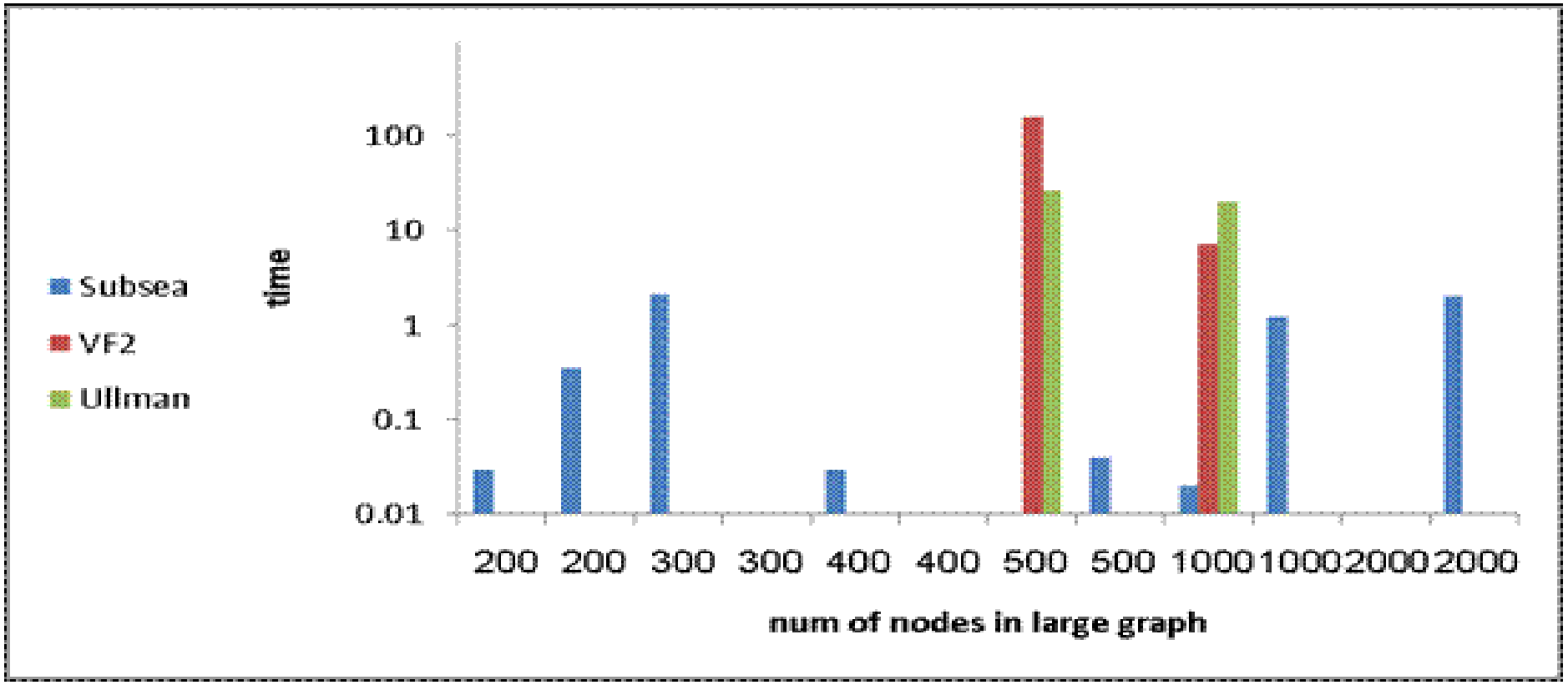
Divide vertices of a given “large” target graph into two parts using the bisection methods.

For each edge with endpoints in distinct parts of the obtained bisection, find the set of all subgraphs (or induced subgraphs) containing this edge and isomorphic to a given pattern graph. (note the edge will start the respective traverse history)

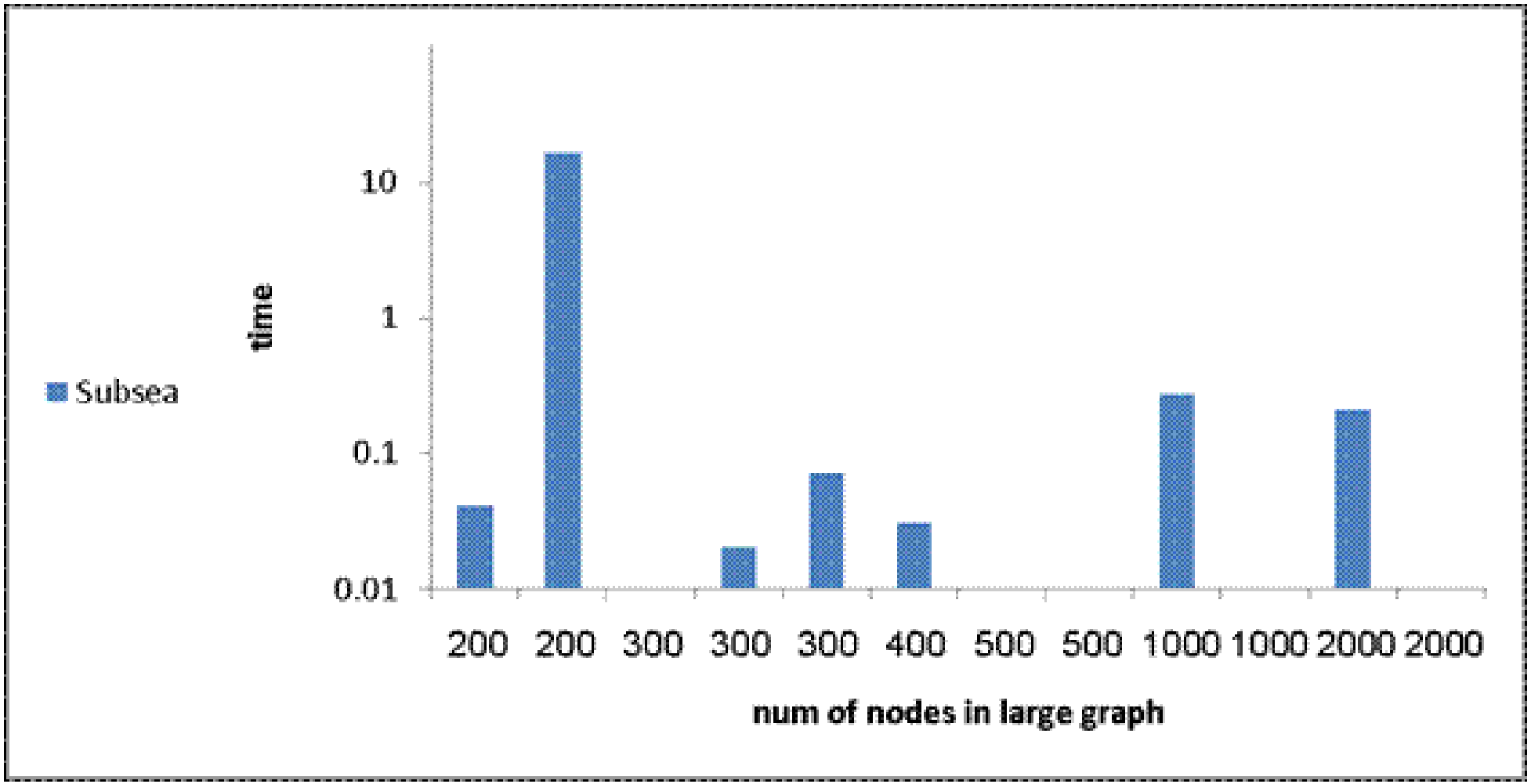
After performing these steps, we continue to apply, in recursive manner, the same approach on the two subgraphs of G induced by the two parts of bisection. We stop when we get a graph with fewer vertices than the pattern graph.

Comparison of Subsea with Ullman's algorithm and Cordella's

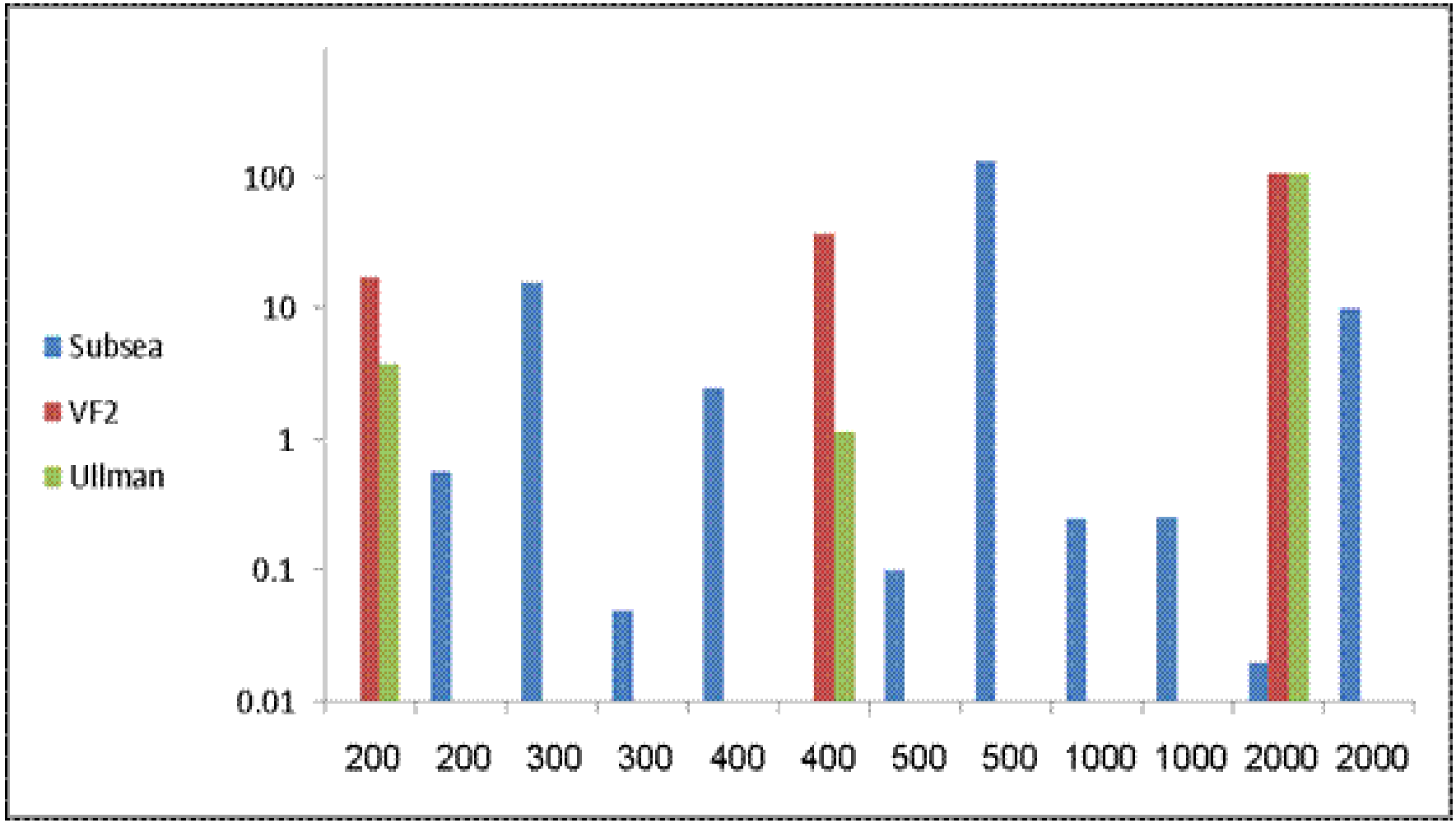
- Subgraph with 15 nodes and 10 labels, uniform label distribution.



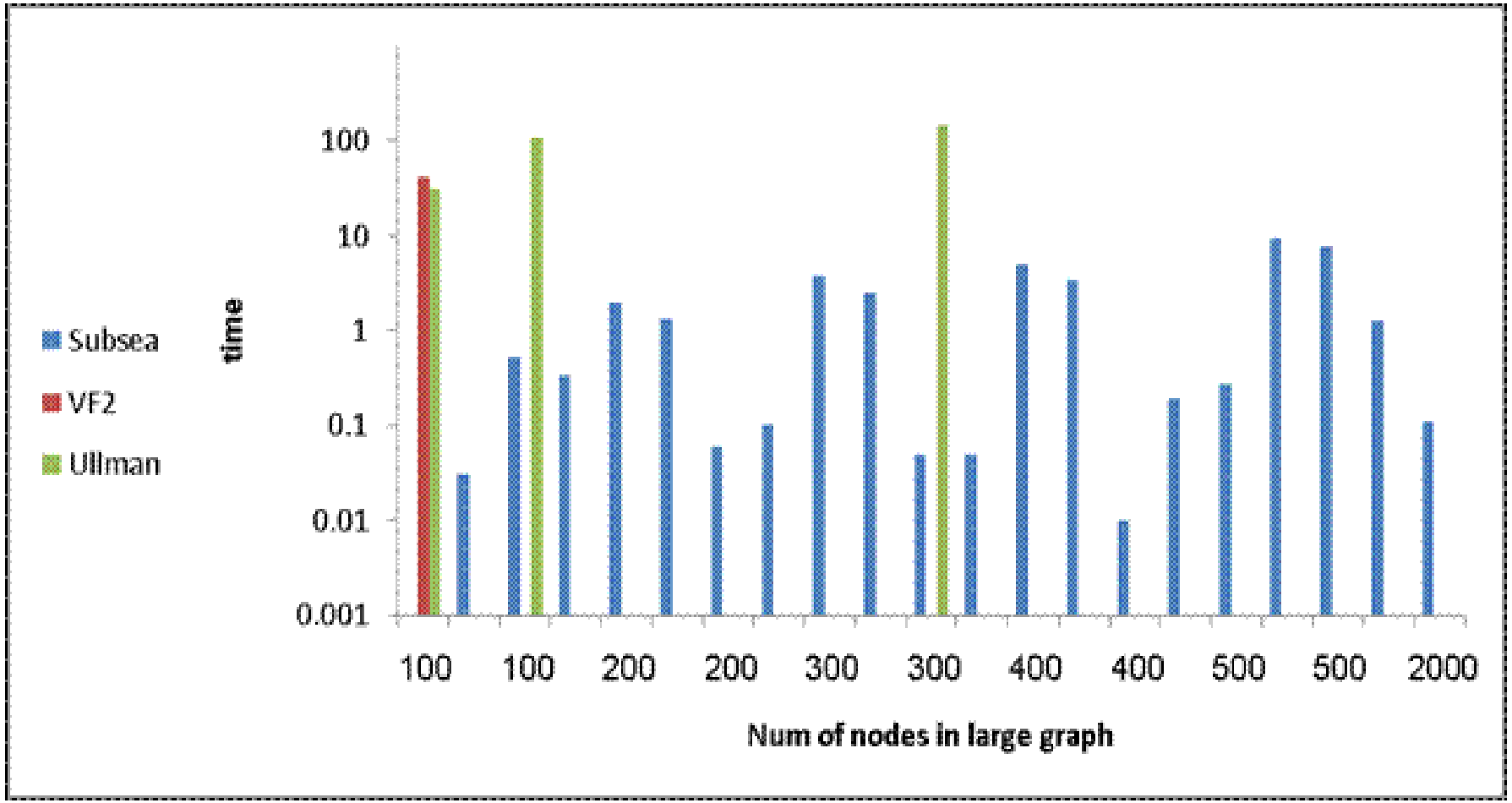
Subgraph with 100 nodes and 10 labels, uniform label distribution



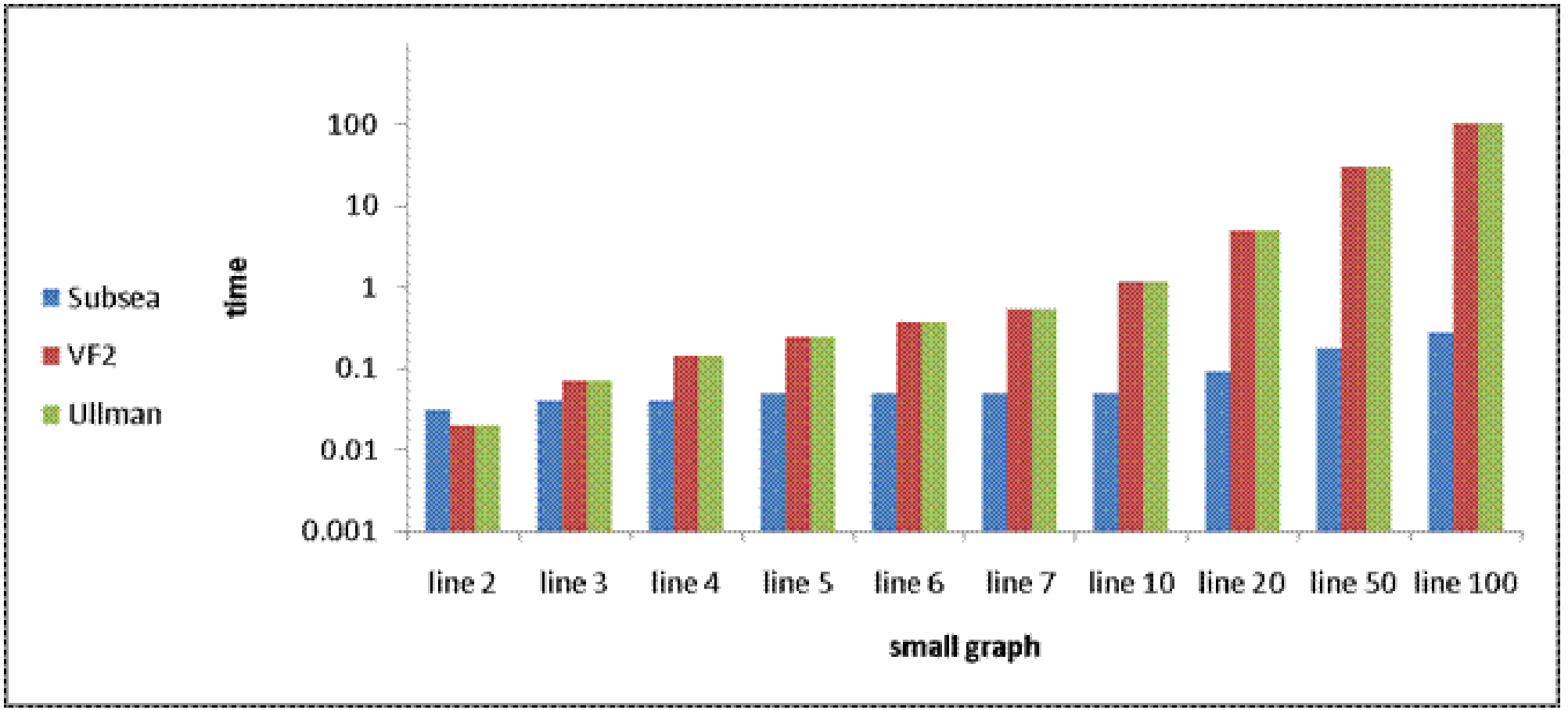
Subgraph on 15 nodes and 5 labels, uniform label distribution.

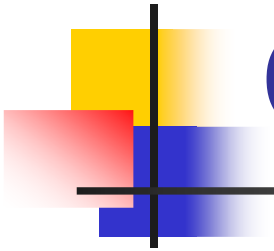


Subgraph of 50 nodes, normal label distribution.



The database graph is an unlabeled line with 200 nodes.





Conclusions

- Subsea was much better than the other two algorithms, especially when multiple occurrences of isomorphism were searched for
- The reason is that each part of the bisection is test independently, and the search is not repeated
- For a single occurrence, the other two algorithms are sometimes better

- Therefore, for a **single graph setting** – choose **Subsea**!

Course Outline (Cont.)

- Searching Graphs and Related algorithms
  - Sub-graph isomorphism (Sub-sea)
  - Indexing and Searching – graph indexing
  - A new sequence mining algorithm
- Web mining and other applications
  - Document classification
  - Web mining
  - Short student presentation on their projects/papers
- Conclusions