# Graph Query Processing

# When it comes to Graphs

- Semi-structured
  - No schema
  - No constraints yet
- No standard query languages
  - A variety of queries used in practice
  - Nontrivial
- What is the complexity of the following problems?
  - Subgraph isomorphism
  - Simple path: given a graph G, a pair (s, t) of nodes in G, and a regular expression R, it is to decide whether there exists a simple path from s to t that satisfies R.
- Query optimization techniques, indexing, updates, …

# Basic Graph Queries And Algorithms

- Graph search (traversal)
- PageRank
- Nearest neighbors
- Keyword search
- Graph pattern matching (a full treatment of itself)

# Path Query

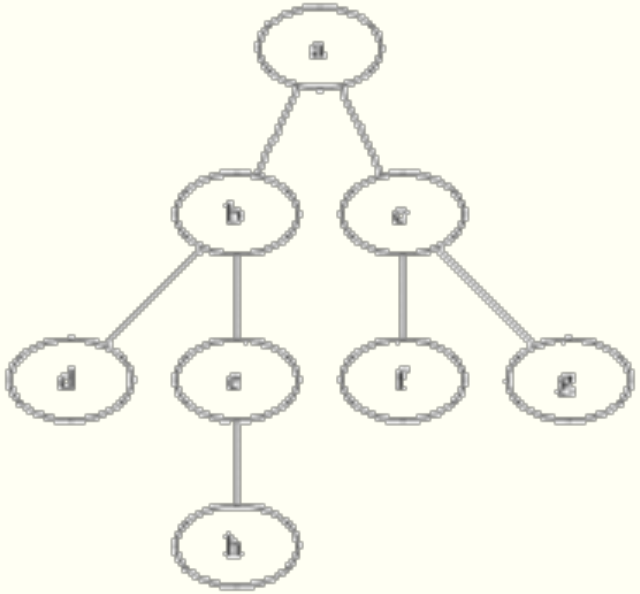
- Reachability
  - Input: A directed graph G and a pair of nodes s and t in G
  - Question: Does there exist a path from s to t in G?
- Distance
  - Input: A directed weighted graph G, and a node s in G
  - Output: The length of shortest paths from s to all nodes in G
- Regular Path
  - Input: A node-labeled directed graph G, a pair of nodes s and t in G, and a regular expression R
  - Question: Does there exist a (simple) path p from s to t that satisfies R?

# Reachability Queries

- Reachability
  - Input: A directed graph G and a pair of nodes s and t in G
  - Question: Does there exist a path from s to t in G?
- Application: (a routine operation)
  - Social graph: Are two people related for security reasons?
  - Biological Networks: find genes that are (directly or indirectly) influenced by a given molecule
    - Nodes: Molecules, reations or physical interactions
    - Edge: Interactions

# Breadth-first Search

- BFS (G, s, t):
  - Let Q be a queue
  - Q.enqueue(s)
  - While Q is not empty
    - v = Q.dequeue()
    - If v is the goal (i.e. t), return True
    - For all edges from v to w in G.adjacentEdges(v) do
      - If w is not labelled as discovered:
        - Label w as discovered
        - w.parent = v
        - Q.enqueue(w)
  - Return false



# BFS Complexity

- BFS (G, s, t):
  - Let Q be a queue
  - Q.enqueue(s)
  - While Q is not empty
    - v = Q.dequeue()
    - If v is the goal (i.e. t), return True
    - For all edges from v to w in G.adjacentEdges(v) do
      - If w is not labelled as discovered:
        - Label w as discovered
        - w.parent = v
        - Q.enqueue(w)
  - Return false

Class: NL-Complete

Complexity:

Space: $O\left(|V| + |E|\right)$

Time: $O\left(\underline{|V|} + \underline{|E|}\right)$

If you want to review complexity theory, here is a video
https://www.youtube.com/watch?v=ZADqzLRDIOQ

# 2-Hop Covers: Strike a balance

- 2 Hop Labels
  - Let $G = (V, E)$ be a directed graph.
  - A 2-hop reachability labeling of $G$ assigns to each vector $v \in V$ a label
  $L(v) = \left(L_{in}(v),\ L_{out}(v)\right)$
  - such that $L_{in}(v),\ L_{out}(v) \subseteq V$ and there is a path from every $x \in L_{in}(v)$ to $v$ and from $v$ to every $x \in L_{out}(v)$.

- Thus, node $\cancel{s}$ u can reach node $v$ iff.

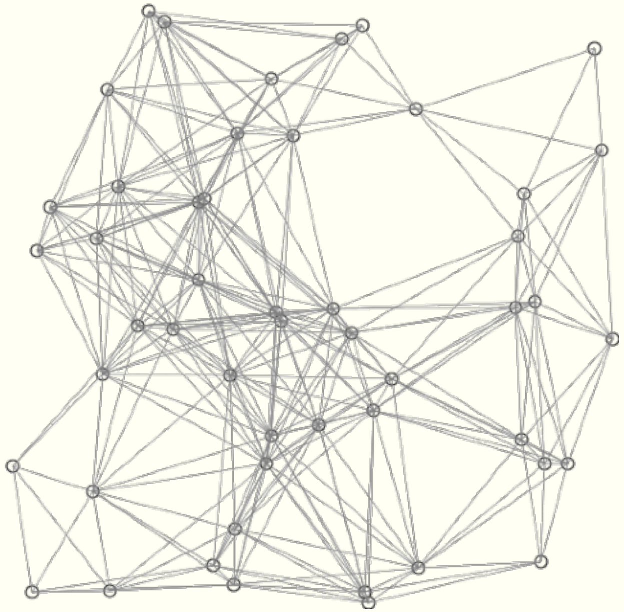
$$L_{out}(u) \cap L_{in}(v) \neq \phi$$

- Testing:
  - Better than $O\left(\left|V\right| + \left|E\right|\right)$
  - Space: $O\left(\left|V\right| \cdot \left|E\right|^{\frac{1}{2}}\right)$

# Distance Queries

- Distance
  - Input: A directed weighted graph G, and a node s in G
  - Output: The length of shortest paths from s to all nodes in G
- Application: Transportation Networks

# Distance Queries

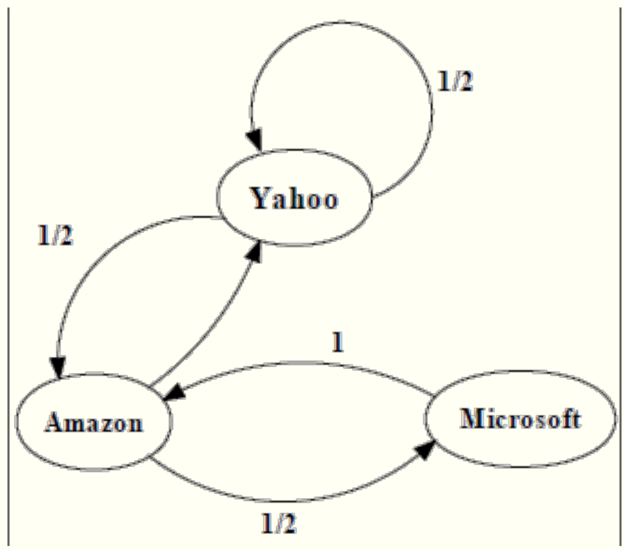


- Dijkstra (G, s, w):
  - Create vertex set Q
  - For each vertex $v \in V$
    - $dist[v] = \infty$
    - $prev[v] = undefined$
    - Q.add(v)
  - $dist[s] = 0$
  - While Q is not empty:
    - u = vertex in Q with minimum distance $dist[u]$
    - Remove u from Q
    - Update the distance of each neighbor $v$ of u to (if it is smaller) $dist[v] = dist[u] + w(u, v)$
- Complexity: If Q is a list, $O\left(|E| + |V|^2\right)$
- Complexity with the right data structure: $O\left(|E| + |V|\log|V|\right)$

# Page Rank

- To Measure the "Quality" of a web page
  - Input: A directed graph $G$ modeling the web, in which nodes represent Web pages, and edges indicate hyperlinks
  - Output: For each node $v$ in graph $G$, $P(v)$ is the likelihood that a random walk over $G$ will arrive at $v$
- Intuition: How a random walk can reach $v$
  - The more pages link to v
  - The more popular those pages that link to v
  - Then, v has a higher chance to be visited

# An example of Simplified PageRank

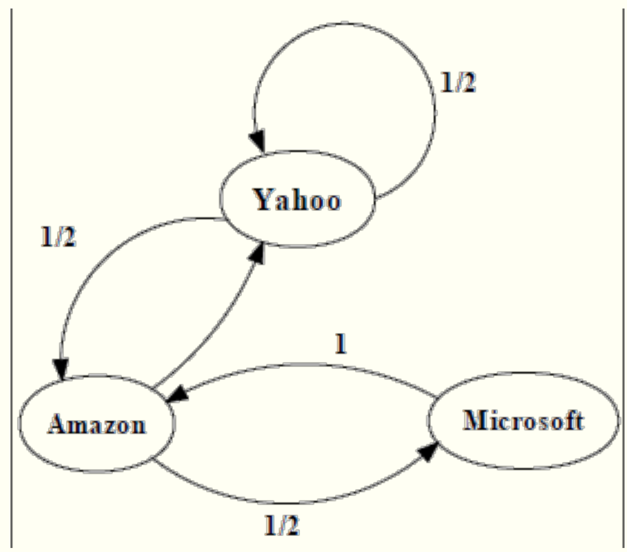



$$M = \begin{array}{cc} & \begin{matrix} Y & A & M \end{matrix} \\ \begin{matrix} Y \\ \phantom{0} \\ \phantom{0} \end{matrix} & \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix} \end{array}$$

$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$\begin{bmatrix} 1/3 \\ 1/2 \\ 1/6 \end{bmatrix} = \underset{M}{\begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix}} \underset{X}{\begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}}$$

# An example of Simplified PageRank (Cont'd)

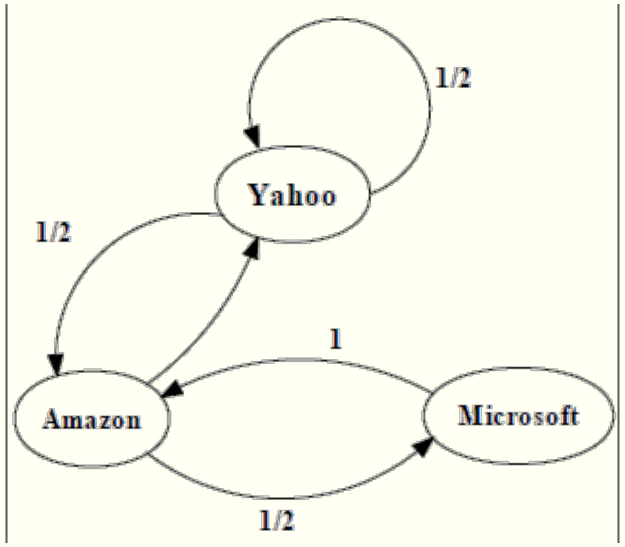



$$M = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix}$$

$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$\begin{bmatrix} 5/12 \\ 1/3 \\ 1/4 \end{bmatrix} = \underbrace{\begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix}}_{M} \begin{bmatrix} 1/3 \\ 1/2 \\ 1/6 \end{bmatrix}$$

# An example of Simplified PageRank (Cont'd)

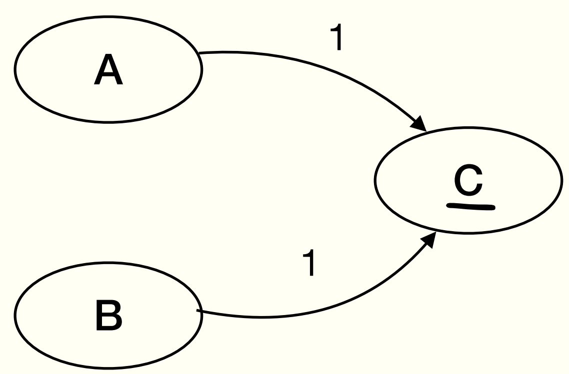



$$M = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix}$$

$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$\begin{bmatrix} 3/8 \\ 11/24 \\ 1/6 \end{bmatrix} \begin{bmatrix} 5/12 \\ 17/48 \\ 11/48 \end{bmatrix} \ \dots \ \begin{bmatrix} 2/5 \\ 2/5 \\ 1/5 \end{bmatrix}$$

# Example of Simplified PageRank



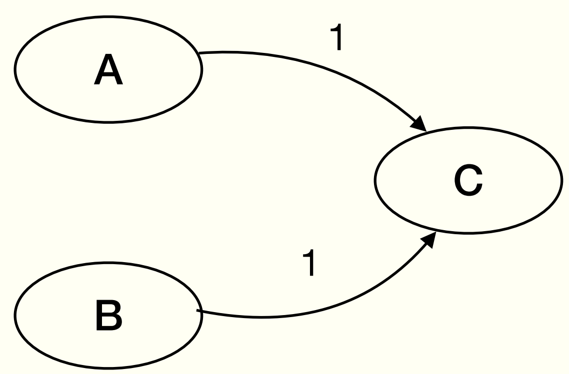

$$M = \begin{array}{c} A \\ \\ C \end{array} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$\begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 2/3 \end{bmatrix}$$

$$\begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 2/3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

# Example of PageRank - Damping Factor




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

$$\begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$\begin{bmatrix} A \\ B \\ C \end{bmatrix} = (1-\alpha) \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix} + \alpha \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$(1-\alpha)$: Damping factor

# Page Rank: (Cont'd)

- Random jump to $v$: $\alpha\left(\frac{1}{|V|}\right)$
  - i.e. The chance of hitting node $v$ among all pages
  - $\alpha$: random jump factor (teleportation factor)
- Hyperlink to $v$: $(1-\alpha)\sum_{u \in L(v)} P(u)/C(u)$
  - $(1-\alpha)$: Damping factor
  - $L(v)$: The set of page that link to $v$
  - $C(u)$: The out-degree of node $u$ (the number of links on $u$)
  - $P(u)$: The probability of u being visited itself
  - $\sum_{u \in L(v)} P(u)/C(u)$: The chances of one to click a hyperlink at a page $u$ and reach $v$

# Page Rank: (Cont'd)

- According to intuition, the likelihood that a page v is visited by a random walk:

$$\alpha\left(\frac{1}{|V|}\right) + (1-\alpha)\sum_{u \in L(v)} P(u)/C(u)$$

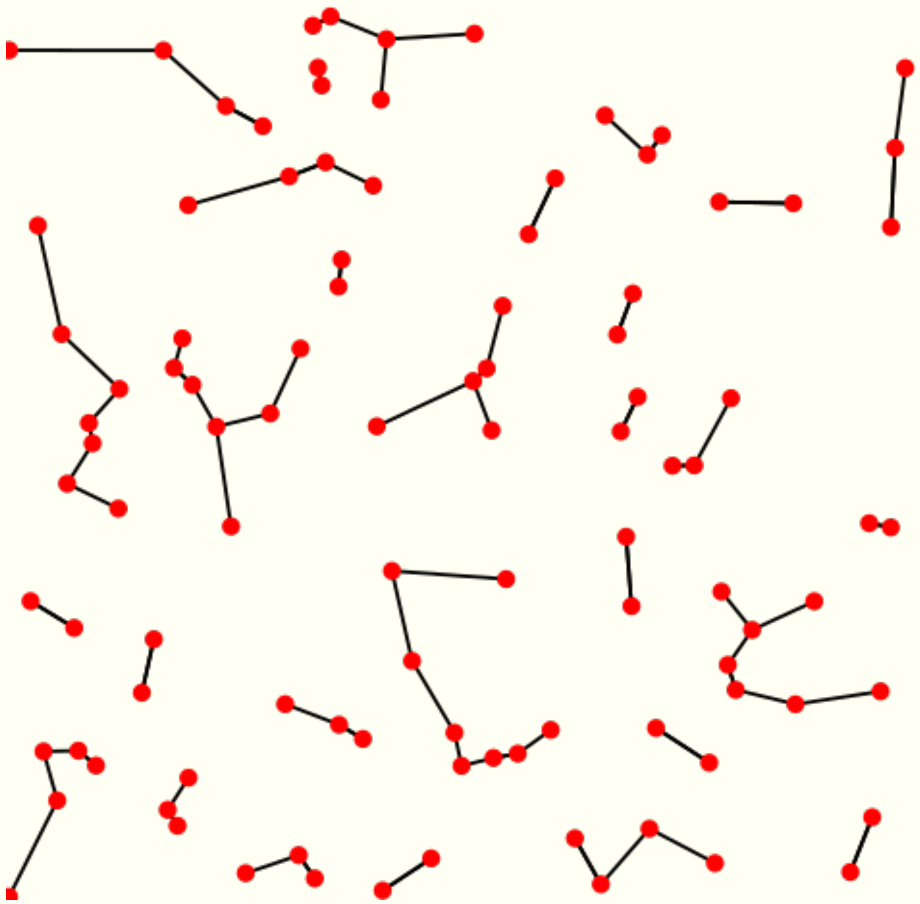
- Recursive computation: For each page $v \in V(G)$,
  - Compute $P(v)$ by using $P(u)$ for all $u \in L(v)$
- Until
  - Converge: no change to any $P(v)$
  - After a fixed number of iterations

# K-Nearest Neighbor

- Nearest neighbor (kNN)
  - Input: A set $S$ of points in a space $M$, a query point $p$ in $M$, a distance function $dist(u, v)$, and a positive integer $k$
  - Output: Find top-k points in $S$ that are closest to p based on $dist(p, u)$
  - Note: The distance can be Euclidean distance, hamming distance, Cosine distance, etc.
- Applications
  - POI recommendation: Find me top-k restaurants close to where I am
  - Classification: classify an object based on its nearest neighbors
  - Regression: property value as the average of the values of its k nearest neighbors
- Methods:
  - Linear search, space partitioning, locality sensitive hashing, compression/clustering based search

# K-Nearest Neighbor Graph

- Graph constructed based on *k*-nearest neighbor nodes
- Node *p* is connected to node *q* if *dist(p, q)* is among the *k*-smallest distances of node p to all other nodes.
- NNG (Nearest Neighbor Graph) is a special case of *k*-NNG with *k*=1



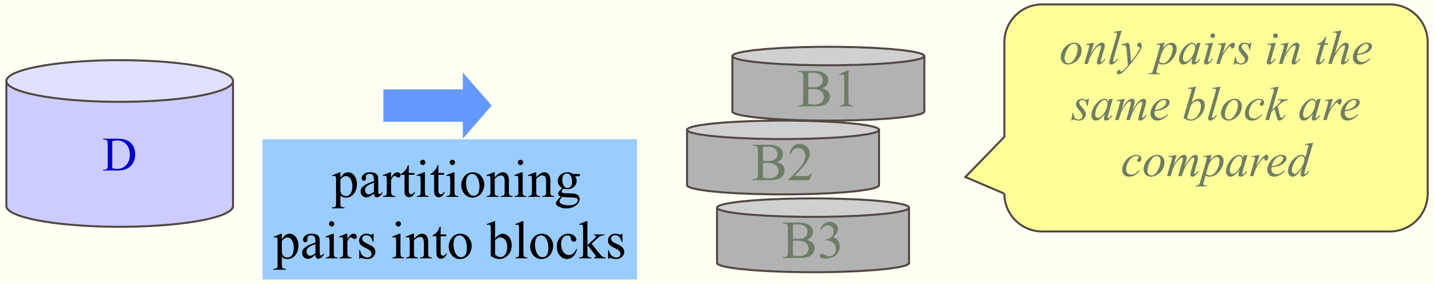
Nearest neighbor graph. (2022, April 24). In *Wikipedia*. https://en.wikipedia.org/wiki/Nearest_neighbor_graph

# kNN Join

- Input: Two datasets $R$ and $S$, a distance function $dist(R, S)$, and a positive integer k
- Output: $pairs\ (r, s)$, for all $r \in R$ and $s \in S$, and is one of the k-nearest neighbors of $r$.
- A naïve algorithm
  - Scanning $S$ once for each object in $R$
  - $O\left(|R| \cdot |S|\right)$: expensive when both datastes are large

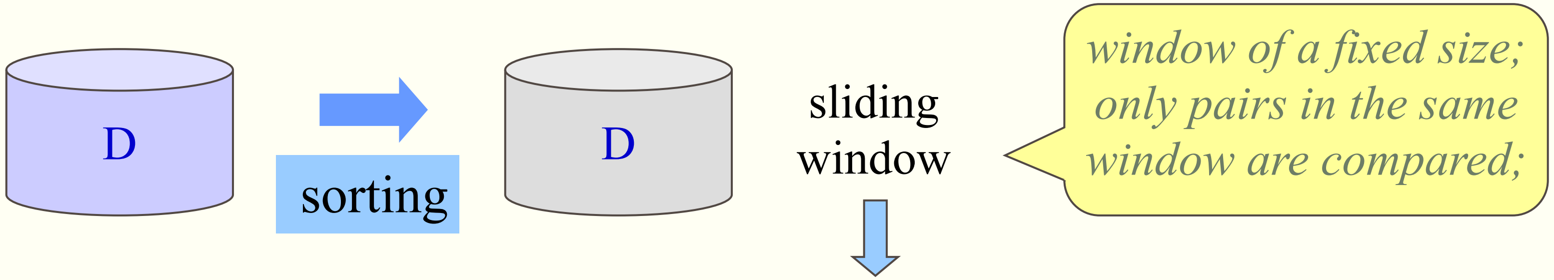
# Blocking and Windowing

- Blocking




GORDER: An Efficient Method for KNN Join Processing. VLDB 2004.

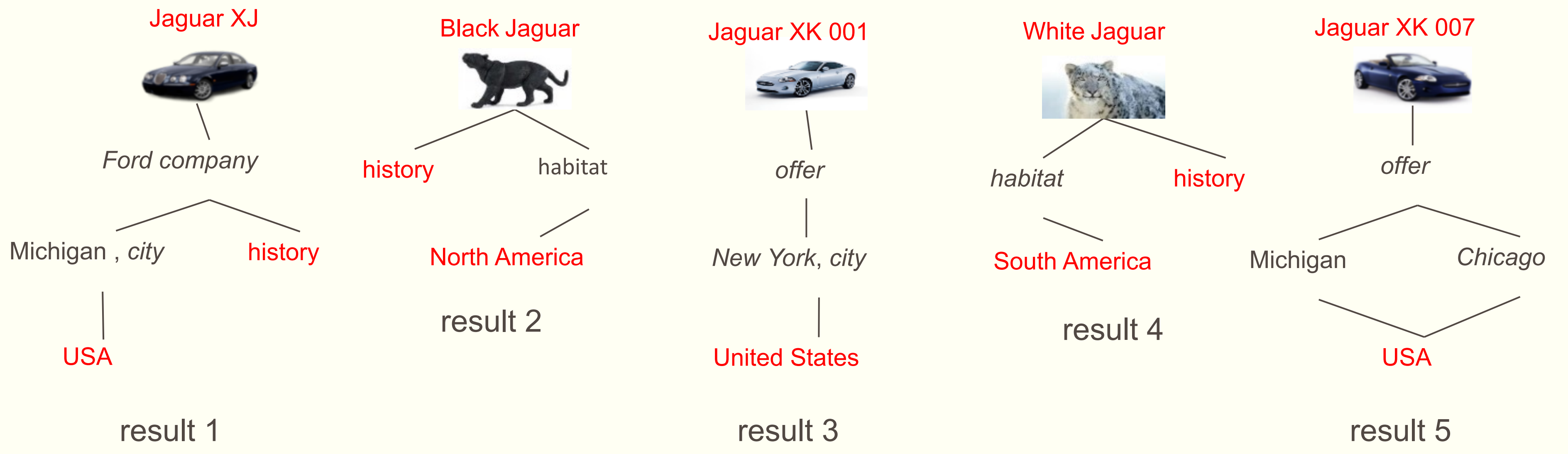
- Windowing



# Keyword Search

- Input: A list $Q$ of keywords, a graph $G$, a positive integer $k$
- Output: top-k "matches" of $Q$ in $G$
- Example: Query: ['Jagur', 'America', 'history']

✓ What makes a match?
✓ How to sort the matches?
✓ How to efficiently find top-k matches?



# Keyword Search: Steiner Tree (Semantics)

- Input: A list $Q$ of keywords, a graph $G$, a weight function $w(e)$ on the edges on $G$, and a positive integer $k$
- Output: top-k Steiner trees that match $Q$
- Match: a subtree $T$ of $G$ such that
  - Each keyword in $Q$ is contained in a leaf of $T$
- Ranking:
  - The total weight of $T$ (the sum of $w(e)$ for all edges $e$ in $T$)
- Complexity:
  - NP-Complete

# Semantics: distinct-root (tree)

- Input: A list $Q$ of keywords, a graph $G$, a weight function $w(e)$ on the edges on $G$, and a positive integer $k$
- Output: top-k distinct trees that match $Q$
- Match: a subtree $T$ of $G$ such that
  - Each keyword in $Q$ is contained in a leaf of $T$
- Ranking:
  - Dist(r,q): from the root of T to a leaf q
  - The sum of distances from the root to all leaves of T
- Diversification:
  - Each match in the top-k answer has a distinct root
- Complexity:
  - $O\Big(|Q|\big(|V|\log|V| + |E|\big)\Big)$

# Semantics: Steiner graphs

- Input: A list $Q$ of keywords, an undirected (unweighted) graph $G$, a positive integer $r$, and a positive integer $k$
- Output: find all $r$-radius Steiner graphs that match $Q$
- Match: a subgraph $G'$ of $G$ such that it is
  - r-radius: the shortest distance between any pair of nodes in G is at most r (at least one pair with the distance)
  - Each key word is contained either in a content node (containing the key word) or a Steiner node (on a simple path between a pair of content nodes)
- Computation: $M^r$, the r-th power of adjacency graph of $G$

# Answering Keyword Queries

- A host of techniques
  - Backward search
  - Bidirectional search
  - Bi-level indexing

- References:

  - G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. ICDE 2002.
  - V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. VLDB 2005.
  - H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. SIGMOD 2007.

# Reading List

- SoQL: an SQL-like language to retrieve paths
- CRPQ: extending conjunctive queries with regular path expressions
  - R. Ronen and O. Shmueli. SoQL: A language for querying and creating data in social networks. ICDE, 2009.
  - P. Barceló, C. A. Hurtado, L. Libkin, and P. T. Wood. Expressive languages for path queries over graph-structured data. In PODS, 2010
- SPARQL: for RDF data
  - http://www.w3.org/TR/rdf-sparql-query/

- Unfortunately, no “standard” query language for graphs, yet