Informed Search for Regular Path Query Reachability

Diego Rivera Correa
Northeastern University
Boston, MA, USA
rivera.di@northeastern.edu

Laurent Bindschaedler
MPI-SWS
Saarbrücken, Germany
bindsch@mpi-sws.org

Abstract

Regular Path Queries (RPQs) are evaluated by traversing a product graph formed from the data graph and a query automaton. While traversal strategies like BFS and DFS have been studied, no work has examined heuristic-guided search for in-memory RPQ evaluation. We design three heuristics of increasing graph-structural richness and evaluate them under a unified parameterization spanning greedy best-first to A*. On LDBC Social Network graphs at two scales, we find that the search regime parameter α dominates heuristic choice: greedy search achieves substantially more pruning over BFS (from 76 to 99%), while A* collapses to BFS-like behavior with overhead. DFS remains fastest for Boolean reachability, but our analysis reveals the structural reasons and identifies conditions under which informed search may prove more valuable.

CCS Concepts

• **Information systems** → **Query languages**; **Graph and subgraph query processing**; • **Theory of computation** → *Graph algorithms analysis*; • **Computing methodologies** → *Heuristic function construction*.

Keywords

regular path queries, graph databases, informed search, A* search, heuristic search, product graph traversal

ACM Reference Format:
Diego Rivera Correa and Laurent Bindschaedler. 2026. Informed Search for Regular Path Query Reachability. In *9th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA '26), May 31-June 05, 2026, Bengaluru, India.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3810460.3812776

1 Introduction

Graph query languages such as SPARQL, Cypher [11], SQL/PGQ [1], and GQL [10] include path-querying constructs for expressing recursive relationships over labeled graphs [2, 6, 7]. A standard way to evaluate such queries is to compile the path expression into an automaton and traverse the corresponding product graph [20]. Making this traversal efficient remains a central challenge in graph query processing [5, 12].

The traversal strategy used for this product-graph walk has direct performance implications. Tetzel et al. [25] showed that DFS and BFS are extreme points in a continuous design space, with DFS dominating in memory for recursive queries. Tetzel et al. [26] demonstrated that data structure specialization, in particular vertically partitioned CSR, bitset-based cycle detection, and state-grouped intermediate results, among others, yields order-of-magnitude improvements in both time and memory. Koschmieder and Leser [17] exploited rare labels as decomposition waypoints for bidirectional BFS, splitting large queries into smaller subproblems. Even the recent output-sensitive algorithm of Abo Khamis et al. [16], which uses degree-aware adaptive BFS, still relies on structural ordering. In these approaches, frontier ordering is driven primarily by structural policies such as BFS, DFS, bidirectional expansion, or decomposition [3, 22, 27, 28], rather than by an explicit heuristic estimate of remaining distance to acceptance. This paper asks: **can informed search reduce the cost of RPQ evaluation in the in-memory setting?**

The gap. In AI search, the move from uninformed to informed search (greedy best-first, A*) routinely reduces explored states by orders of magnitude [23]. Product-graph traversal naturally induces a search problem whose states are pairs (v, q) and whose goals are product states with accepting automaton components. This suggests importing heuristic-guided search techniques such as greedy best-first search and A* [15]. To our knowledge, the only prior use of A* for RPQs is by Baier et al. [4], who used A* for RPQs over Web/Linked Data. Their setting is dominated by network latency (~1 s per step), where even a weak heuristic provides large gains. The in-memory setting is fundamentally different: state expansion is much cheaper, so any benefit from heuristic guidance must outweigh priority-queue maintenance and heuristic-computation overhead. Whether this is achievable, and under which query objectives, has not been studied.

Vision: cost-based traversal selection. Although there is a rich theoretical literature on RPQs, implementing these concepts into real-life systems is still work in progress [21]. We view this work as a first step toward systematic traversal strategy selection for RPQs, analogous to join ordering in relational query optimization. This is increasingly urgent: GQL alone defines 27 path-finding modes (`ANY`, `SHORTEST`, `ALL SHORTEST`, etc.) [9], each placing different demands on the traversal engine. The long-term goal is a cost model that, given a query objective (existence, shortest, enumeration), automaton structure, and graph statistics, selects among DFS, BFS, bidirectional search, and informed search. This paper contributes the first empirical characterization of where informed search sits in this design space, focusing on the simplest objective: Boolean reachability.

Scope. We focus on single-source Boolean reachability with a fixed start vertex. Boolean reachability is the query objective underlying SPARQL `ASK` queries, access-control checks ("can user u reach resource r via a chain of delegations matching pattern R?"), and

This work is licensed under a Creative Commons Attribution 4.0 International License.
GRADES-NDA '26, Bengaluru, India
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2695-8/2026/05
https://doi.org/10.1145/3810460.3812776

constraint validation in knowledge graphs. It is also the most favorable setting for informed search, since it admits early termination upon finding any accepting state. We defer shortest-path objectives, bidirectional search, weighted edges, and multi-source evaluation to ongoing work. Our goal is to determine whether informed search merits further investigation for in-memory RPQ evaluation and to establish the methodology and cost model for studying richer objectives.

Contributions. This paper makes three contributions:

(1) We initiate the study of informed search for in-memory RPQ evaluation. While prior work has explored uninformed traversal strategies (BFS, DFS, and parameterized variants) and applied A* in latency-dominated Web settings, no work has examined whether heuristic guidance can reduce traversal cost when the graph resides entirely in memory.
(2) We analyze three heuristics of increasing graph-structural richness: Minimum Automaton Distance (MAD), which lower-bounds remaining cost using only the query automaton; Outgoing Transition Degree (OTD), which refines MAD with local edge-label topology; and Label Reachability Score (LRS), which detects the absence of mandatory labels in the vertex neighborhood.
(3) We evaluate these heuristics on LDBC Social Network Benchmark graphs under the Boolean reachability objective, characterizing the conditions under which informed search reduces states expanded relative to BFS and DFS, and identifying the overhead profile that determines when the reduction translates to wall-clock improvement.

2 Background and Problem Setting

We use standard NFA notation [24]; only the components needed for product-graph traversal are introduced here.

2.1 RPQ Evaluation via Product Graphs

Let $G = (V, E, \Sigma, \lambda)$ be a labeled directed graph with vertex set V, edge set E, label alphabet Σ, and labeling function $\lambda : E \to \Sigma$. A Regular Path Query is a regular expression R over Σ. Its evaluation constructs a non-deterministic finite automaton $A = (Q, \Sigma, \delta, q_0, F)$ equivalent to R and traverses the *product graph* $G \times A$.

Example 1. *Consider the graph G in Figure 1. The set of edge labels Σ is equal to* {`likes`, `hasCreator`, `hasTag`, `livesIn`}. *Similarly,* $\lambda(v_0, v_4) =$ `likes`.

Product graph. States of the product graph are pairs $(v, q) \in V \times Q$. A transition $(v, q) \to (v', q')$ exists when there is an edge $(v, v') \in E$ with $\lambda(v, v') = a$ and $q' \in \delta(q, a)$. The product graph is not materialized; it is constructed on-the-fly during traversal.

Boolean reachability. Given a start vertex $s \in V$, does there exist any state (v, q_f) with $q_f \in F$ reachable from (s, q_0) in the product graph? If so, at least one path exists in G starting at s whose edge-label sequence is accepted by A.

Query objectives. RPQ evaluation serves three objectives with different traversal demands: *Boolean reachability* (does any conforming path exist?), which admits early termination on the first accepting state; *shortest path* (find a minimum-cost conforming path), which requires optimality guarantees from the search; and *enumeration* (return all conforming paths), which requires full exploration with no early termination possible. Boolean reachability is the most favorable setting for informed search, since early termination is the primary performance lever. This paper begins here.

2.2 The Informed Search Spectrum

We unify traversal strategies under a single parameterized priority function:

$$f(s) = \alpha \cdot g(s) + (1 - \alpha) \cdot h(s), \quad \alpha \in [0, 1] \tag{1}$$

where $g(s)$ is the number of product-graph transitions from the source (i.e., depth under unit edge costs) and $h(s)$ a heuristic estimate of remaining cost. Setting $\alpha = 1$ with $h \equiv 0$ recovers BFS expansion order (under unit costs); $\alpha = 0$ gives greedy best-first; $\alpha = 0.5$ with admissible h gives A* [15, 23] (order-equivalent to the standard $f = g + h$). The search maintains a closed set over product states (v, q) to avoid duplicate expansion. For Boolean reachability, low α is well-motivated since any path suffices (termination dominates optimality).

3 Graph-Structural Heuristics

We design three heuristics of increasing graph-structural richness. Furthermore, each is usable within the unified priority function of Equation (1). Admissibility guarantees that A* ($\alpha = 0.5$) finds optimal paths; for Boolean reachability it is an efficiency property rather than a correctness requirement, since any complete search will eventually find a solution if one exists.

Example 2 (Content-creator chain query). *Consider querying a social network for cities reachable through chains of liked content and their creators. Starting from a Person, the query follows a* `likes` *edge to a Post, then* `hasCreator` *back to the Post's author (a Person). This likes/hasCreator cycle may repeat: the author likes another Post, which has its own creator, and so on. Eventually, the chain exits through a* `livesIn` *edge to a City, reaching acceptance.*

Figure 2 shows the NFA for this query. State q_2 is the critical decision point: the search may either continue the cycle (via `likes` *back to q_1) or exit toward acceptance (via* `livesIn` *to q_3). This creates a natural setting for comparing heuristic guidance.*

Suppose we begin traversal from (v_0, q_0). After expanding v_0's neighbors, we reach three successor instances: $(v_1, q_1), (v_3, q_1), (v_4, q_1)$. Which instance should be processed first? An uninformed search (BFS or DFS) treats the three states equally: they are at the same depth and the same NFA state. The three heuristics below differ in how (and whether) they detect which state deserves higher priority.

3.1 H1: Minimum Automaton Distance (MAD)

This heuristic was introduced by Baier et al. [4] for A* search over Web/Linked Data RPQs. We adopt it as our baseline heuristic and build OTD and LRS as refinements that incorporate graph-structural information absent from MAD. Baier et al. applied MAD in a network-latency-dominated setting where per-step costs are milliseconds; we study whether it remains effective when steps cost nanoseconds.

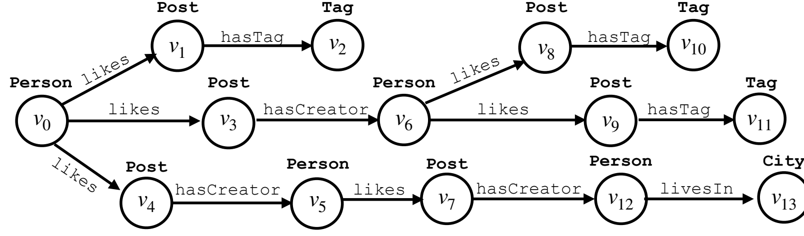

Figure 1: Labeled directed graph G over a social-network schema with four edge types.

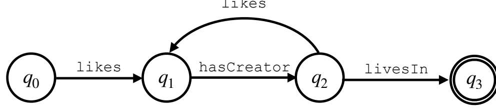

Figure 2: NFA for $(\texttt{likes/hasCreator})^+ \cdot \texttt{livesIn}$.

Intuition. The NFA encodes how much of the regular expression remains to be matched. Regardless of where the search stands in the data graph, any completing path must traverse at least as many automaton transitions as the shortest path from the current NFA state to any accepting state. MAD uses this as a progress meter, a lower bound on remaining work derived entirely from the query structure. More formally:

$$h_{\text{MAD}}(v, q) = \min_{q_f \in F} d_A(q, q_f) \tag{2}$$

where $d_A(q, q_f)$ is the shortest-path distance in the NFA from state q to accepting state q_f.

Admissibility. Any path from a product-graph state (v, q) to an accepting state (v', q_f) with $q_f \in F$ must traverse a sequence of product-graph edges, each of which advances the NFA by exactly one transition. The number of such transitions is at least $d_A(q, q_f)$, the shortest-path distance in the NFA from q to q_f. Under unit edge costs, each transition corresponds to one data-graph edge, so $h_{\text{MAD}}(v, q) = \min_{q_f \in F} d_A(q, q_f) \le d^*$, where d^* is the true minimum remaining cost. MAD is therefore admissible.

Example 3 (MAD's "Uniform" View). *Recall that if we evaluate Example 2 from (v_0, q_0), we are able to reach 3 follow-up instances: $(v_1, q_1), (v_3, q_1), (v_4, q_1)$. Under MAD, all three instances are equally-valued since they are equally-far from an accepting state. Hence, the traversal could begin by processing (v_1, q_1), which is a dead-end, wasting computation.*

Limitation. Two product states (v_1, q) and (v_2, q) receive identical h-values regardless of the local topology at v_1 and v_2. MAD cannot detect dead ends, bottlenecks, or missing labels in the data graph (see Example 3). When many product-graph states share the same NFA state, MAD provides no discrimination among them; the priority queue degenerates to FIFO (BFS) order within each h-level.

Algorithm 1 summarizes how MAD is computed. The precomputation (lines 1–11) is a standard reverse BFS on the NFA, propagating shortest distances from accepting states backward. Because the NFA has very few states ($|Q|$ is typically 2–5 for practical RPQs), this completes in microseconds. At query time (lines 12–13), evaluating MAD for any product-graph state (v, q) requires only reading $d[q]$ from the table.

Algorithm 1: MAD: Minimum Automaton Distance

Input: NFA $A = (Q, \Sigma, \delta, q_0, F)$
Output: Lookup table $d[q]$ for all $q \in Q$
/* Precomputation: run once per query. We perform a multi-source BFS starting from all accepting states on the reversed NFA. This computes, for every NFA state q, the minimum number of transitions needed to reach any accepting state. Since the NFA is small (typically $|Q| \le 5$), this takes negligible time. */
1 **foreach** $q \in Q$ **do**
2 | $d[q] \leftarrow \infty$
3 Initialize empty queue W
4 **foreach** $q_f \in F$ **do**
5 | $d[q_f] \leftarrow 0$
6 | W.enqueue(q_f)
7 **while** W *is not empty* **do**
8 | $q \leftarrow W$.dequeue$()$
9 | **foreach** *state q' that has a transition* into q *in the NFA* **do**
10 | | **if** $d[q']$ *is* ∞ **then**
11 | | | $d[q'] \leftarrow d[q] + 1$
12 | | | W.enqueue(q')
/* Evaluation: called once per product-graph state. Simply returns the precomputed distance. This is a single array lookup costing $O(1)$. */
13 **Function** h_{MAD}(*vertex v, NFA state q*)**:**
14 | **return** $d[q]$

3.2 H2: Outgoing Transition Degree (OTD)

Intuition. Among states at the same automaton distance from acceptance, vertices differ in how many of their outgoing edges carry labels that match the next required NFA transition. A vertex where many edges match offers more continuation paths and is less likely to be a dead end. A vertex with few or no matching edges will force the search to backtrack. OTD formalizes this observation: between two equally-progressed states, prioritize the one with more options. More formally:

$$h_{\text{OTD}}(v, q) = h_{\text{MAD}}(v, q) - \gamma \cdot \text{MR}(v, q) \tag{3}$$

$$\text{MR}(v, q) = \frac{|\{e \in \text{out}(v) : \lambda(e) \in \delta(q, \cdot)\}|}{|\text{out}(v)|} \tag{4}$$

where $\text{MR}(v, q)$ is the match ratio: the fraction of v's outgoing edges carrying a label valid under NFA state q. The parameter $\gamma > 0$ controls the weight of the topological signal.

Admissibility. OTD subtracts a non-negative term ($\gamma \cdot \text{MR} \in [0, \gamma]$) from MAD. Since $h_{\text{OTD}} \le h_{\text{MAD}} \le d^*$, OTD is admissible for

all $\gamma > 0$. In practice, we clamp h_{OTD} to zero from below to avoid degenerate priority ordering from negative values.

Computation cost. OTD requires a per-vertex label histogram (a count of outgoing edges per label). This pairs naturally with the vertically partitioned CSR representation: for each label partition, the degree of vertex v is offset$[v+1]$ − offset$[v]$, computed in $O(1)$. The set of valid transition labels $L_q = \{a \in \Sigma : \delta(q,a) \neq \emptyset\}$ is precomputed per NFA state. Per-state evaluation sums $|L_q|$ histogram entries, costing $O(|\Sigma|)$.

Example 4 (OTD's "Greedy" Approach). *Assume that in Example 2 we reach the following two states: (v_6, q_2), (v_5, q_2) (starting from v_0, both instances traversed a* `likes` *and* `hasCreator` *edge). Under OTD, the idea is to prioritize (v_6, q_2) since v_6 has more outgoing edge labels that satisfy the regular expression than v_5. v_6 has 2 outgoing* `likes`*, whereas v_5 only has 1.*

Limitation. OTD is a one-step lookahead: it measures how many of the current vertex's edges can advance the automaton, but it says nothing about what lies beyond those edges. Returning to Example 4, (v_6, q_2) has no `livesIn` edge down the road, but does have many outgoing `likes` edges. OTD sees a high match ratio for the `likes` transition at q_2 and lowers v_6's heuristic value accordingly, treating it as a promising state. Yet every one of those `likes` edges leads to Posts whose creators also lack `livesIn` edges. These are dead ends that the search will only discover after expanding them. In general, OTD can be misled by hub vertices that generate many immediate successors, none of which lead to acceptance.

Algorithm 2 summarizes OTD. The graph precomputation (lines 1–4) builds a per-vertex label histogram. With a vertically partitioned CSR, the degree of vertex v under label ℓ is simply offset$_\ell[v+1]$ − offset$_\ell[v]$. The query precomputation (lines 5–6) collects the valid labels at each NFA state. At evaluation time (lines 7–14), the match ratio is computed by summing the counts for each valid label and dividing by the total degree. The tuning parameter $\gamma \leq 1$ controls how aggressively the topological signal adjusts the MAD baseline.

Algorithm 2: OTD: Outgoing Transition Degree

```
Input: Graph G = (V, E, Σ, λ), NFA A = (Q, Σ, δ, q0, F)
/* Graph precomputation: run once when the graph is
   loaded. For each vertex, we count how many outgoing
   edges carry each label. With vertically partitioned
   CSR, this is simply the difference of two adjacent
   offset values per label partition.                  */
1  foreach vertex v ∈ V do
2      foreach label ℓ ∈ Σ do
3          count[v][ℓ] ← number of outgoing edges from v with
           label ℓ
4      total[v] ← total out-degree of v
/* Query precomputation: run once per query. For each
   NFA state, we collect which labels can advance the
   automaton.                                          */
5  foreach NFA state q ∈ Q do
6      L[q] ← {a ∈ Σ : δ(q, a) ≠ ∅}
/* Evaluation: called once per product-graph state. We
   compute the match ratio: the fraction of v's
   outgoing edges that carry a label the NFA can
   currently accept. A high match ratio means the
   vertex has many ways to continue the pattern, so we
   subtract it from the MAD baseline to give the state
   higher priority.                                    */
7  Function h_OTD(vertex v, NFA state q):
8      matching ← 0
9      foreach label a ∈ L[q] do
10         matching ← matching + count[v][a]
11     if total[v] = 0 then
12         return h_MAD(v, q)
13     else
14         MR ← matching / total[v]
15         return h_MAD(v, q) − γ · MR
```

3.3 H3: Label Reachability Score (LRS)

MAD ignores the data graph entirely; OTD incorporates only immediate neighbors. A natural question is whether looking *deeper* into the graph neighborhood can improve guidance. LRS tests this design principle by checking whether labels that the NFA *requires* on every path to acceptance are present within 2 hops. We include LRS not because we expect it to dominate on all benchmarks, but to probe the conditions under which graph-structural awareness helps and to understand when it does not.

Intuition. If the NFA requires label ℓ on every path from the current state to acceptance, and ℓ does not appear on any edge within 2 hops of the current vertex, then any completing path must first travel to a region of the graph where ℓ exists. LRS detects these structural dead zones before the search expands into them, tightening the lower bound beyond what MAD provides.

Why $k = 2$. At $k = 1$, LRS checks only the immediate outgoing edges of v, which is already discovered during on-the-fly product graph construction at expansion time. The check is redundant with the traversal itself. At $k = 2$, LRS looks one step *ahead* of what the product graph expansion reveals, detecting missing labels in the next frontier before expanding into it. At $k \geq 3$, precomputation cost becomes prohibitive for large graphs ($O(|E| \cdot d_{\text{avg}}^{k-1})$), and the diminishing marginal information does not justify the expense. More formally:

$$h_{\text{LRS}}(v,q) = h_{\text{MAD}}(v,q) + \beta \cdot |M(q) \setminus \text{Labels}_2(v)| \tag{5}$$

where:

- $M(q)$ is the set of labels *mandatory* on every path from q to any accepting state in the NFA. A label ℓ is mandatory at q if every path from q to F in the NFA traverses at least one transition labeled ℓ.
- $\text{Labels}_2(v) = \{\lambda(e) : e \in E(N_2(v))\}$ is the set of distinct edge labels present within the 2-hop neighborhood of v.
- $\beta > 0$ controls the penalty weight.

Admissibility. LRS is not admissible in general. Consider a state (v, q) where two labels $a, b \in M(q)$ are both absent from $\text{Labels}_2(v)$. With $\beta = 1$, LRS adds a penalty of 2 to h_{MAD}. However, a single step beyond the 2-hop boundary may reach a vertex u whose neighborhood contains both a and b-labeled edges; the true extra cost beyond h_{MAD} is then 1, not 2, and LRS overestimates. To recover admissibility, we can cap the penalty: $h_{\text{LRS}}^{\text{cap}}(v,q) = h_{\text{MAD}}(v,q) + \min(1,\ |M(q) \setminus \text{Labels}_2(v)|)$. In our experiments, LRS collapses to

Algorithm 3: LRS: Label Reachability Score

```
Input: Graph G = (V, E, Σ, λ), NFA A = (Q, Σ, δ, q_0, F)
/* Graph precomputation: run once when the graph is
   loaded. For each vertex, we record which edge labels
   exist within 2 hops. First, we collect the labels on
   the vertex's own outgoing edges (1-hop). Then, for
   each neighbor, we add that neighbor's outgoing edge
   labels (2-hop). The result is stored as a bitset of
   size |Σ| per vertex. Total cost: O(|E| · d_avg).      */
1  foreach vertex v ∈ V do
2  |   nearby[v] ← empty bitset of size |Σ|
3  |   foreach edge (v, u) in the graph do
4  |   |   Set bit λ(v, u) in nearby[v]
5  |   foreach edge (v, u) in the graph do
6  |   |   foreach edge (u, w) in the graph do
7  |   |   |   Set bit λ(u, w) in nearby[v]
/* Query precomputation: run once per query. For each
   NFA state, we compute which labels are mandatory: a
   label is mandatory at state q if it appears on
   every possible path from q to an accepting state.
   This is computed by analyzing the NFA structure.  */
8  foreach NFA state q ∈ Q do
9  |   M[q] ← labels that appear on every NFA path from q to any
   |     q_f ∈ F
/* Evaluation: called once per product-graph state. We
   check how many mandatory labels are missing from
   the 2-hop neighborhood. Each missing label means
   the search must travel at least one extra step
   beyond the MAD estimate to reach a region of the
   graph where that label exists.                    */
10 Function h_LRS(vertex v, NFA state q):
11 |   missing ← |M[q] \ nearby[v]|
12 |   return h_MAD(v, q) + β · missing
```

MAD on LDBC regardless of capping, so we report the uncapped variant and note that it serves as a priority signal for greedy search rather than a guaranteed lower bound.

Example 5 (LRS's "Foreshadowing" Approach). *Similar to the situation in Example 4, assume we reach the following two states: $(\underline{v_6}, \underline{q_2})$, $(\underline{v_5}, \underline{q_2})$. Under LRS, the priority is to check whether the labels required to complete the query pattern are entirely absent from a vertex's local neighborhood. Looking at the $(\underline{v_6}, \underline{q_2})$ instance, if we look 2 hops in advance, we notice that it contains* `hasCreator` *and* `hasTag` *edges, which do not advance the NFA to an accepting state. However, we notice that the $(\underline{v_5}, \underline{q_2})$ instance, its 2-hop vicinity includes* `hasCreator` *and* `livesIn`*, which is needed to reach an accepting state. Hence, $(\underline{v_5}, \underline{q_2})$ is prioritized.*

Limitation. The LRS heuristic checks for the *existence* of mandatory labels within 2 hops of the current vertex, but does not enforce ordering constraints. This existence-based design keeps LRS computationally efficient, but it is inherently loose compared to an order-aware heuristic. Future work could strengthen label reachability by tracking *which* mandatory labels have been consumed on the current path (via state augmentation) and only counting future labels as "needed."

Algorithm 3 summarizes LRS. The graph precomputation (lines 1–8) builds the 2-hop label reachability matrix using bitwise OR over neighbors' 1-hop bitsets, costing $O(|E| \cdot d_{\text{avg}})$. This cost is amortized over all queries and excluded from per-query time. The query precomputation (lines 9–10) identifies mandatory labels at each NFA state via reachability checks. At evaluation time (lines 11–12), the missing-label count is a single bitset difference operation. The parameter $\beta = 1$ weights each missing label equally.

4 Experimental Setup

Implementation. Our prototype is implemented in C++ (compiled with GCC 12, `-O3`). All data structures (graph, automaton, heuristic lookup tables, priority queue) reside in memory. Each configuration is executed once per source vertex (approximately 20 runs per query); we report the median and standard deviation of execution time and state count across sources. Heuristic precomputation (label histograms for OTD, 2-hop label matrices for LRS) is performed once per graph and excluded from query time.

Graphs. We use the LDBC Social Network Benchmark [8] at scale factors SF1 (3.1M vertices, 5.7M edges) and SF3 (9.0M vertices, 18M edges). The graph is stored in vertically partitioned CSR [26]: one CSR per edge label, enabling $O(1)$ degree lookup per label. Reverse edges for inverse transitions (ˆ) are stored as separate partitions. The visited structure uses one fixed-size bitset per automaton state, reset via SIMD memset.

Source Vertex Selection. To assess robustness across graph structure variation, we stratify vertices by out-degree into three tiers (low, mid, high) and sample $n = 20$ vertices evenly across tiers.

Queries. We use eight recursive query patterns spanning diverse structural profiles. Q1, Q2, Q5, Q6 are referenced from [25]:

- Q1 `(likes/hasCreator)+`
- Q2 `(likes/ˆlikes/knows)+`
- Q3 `(likes/isLocatedIn/ˆisLocatedIn/ˆworkAt)+`
- Q4 `(knows/hasInterest/ˆhasInterest/knows)+`
- Q5 `knows+`
- Q6 `(hasCreator/likes)+`
- Q7 `(knows/likes/ˆhasCreator)+`
- Q8 `(hasInterest/ˆhasTag/ˆhasCreator)+`

These queries test the hypothesis that heuristic quality improves with label-frequency contrast and mandatory-label selectivity. For each query, we sweep minimum accepted path length $\ell_{\min}$ from 1 to 15; we focus primarily on the transition region $\ell_{\min} \in [3, 10]$ where heuristic guidance becomes more important.

Configurations. DFS and BFS as baselines; Greedy-{MAD, OTD, LRS} (α=0); A*-{MAD, OTD, LRS} (α=0.5); WA*-LRS at $\alpha \in \{0.3, 0.7\}$.

Metrics. Per source vertex: (1) Execution time (median ± standard deviation over 20 sources). (2) States expanded: product graph states popped before the first accepting state. (3) Pruning ratio: fraction of BFS states avoided. (4) Heuristic tightness: mean heuristic value across expansions (h_{mean}), and slack at solution ($h_{\text{slack}} = g^* - h(v^*)$). All metrics except time are aggregated by mean across sources.

5 Results

We organize our findings around three observations that emerged consistently across queries and both scale factors.

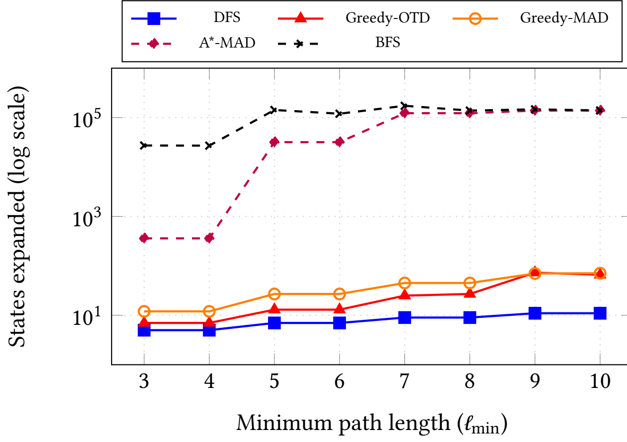

Figure 3: States expanded vs. minimum path length for Q_1 (likes/hasCreator)$^+$ on SF1. DFS and greedy configurations remain orders of magnitude below BFS and A*, which converge at high $\ell_{\min}$.

Table 1: States expanded (mean) and wall-clock time (ms, mean) at representative $\ell_{\min}$ values. SF1: 3.1M vertices, 5.7M edges. SF3: 9.0M vertices, 18.0M edges. Q3, Q7, Q8 are omitted: see text.

		DFS		Gr-MAD		Gr-OTD		A*-MAD		BFS	
Query	ℓ	St	ms	St	ms	St	ms	St	ms	St	ms
SF1											
Q1	7	9	0.4	45	2.1	25	2.0	123K	76	173K	24
Q2	7	12	0.5	3.9K	29	8.6K	53	143K	91	179K	36
Q4	9	25	0.4	12K	9.3	18K	14	36K	20	52K	7.6
Q5	7	56	0.2	57	0.2	30	0.3	3.1K	1.7	3.1K	0.5
Q6	7	9	0.3	2.6K	1.8	1.4K	1.4	125K	68	133K	24
SF3											
Q1	7	9	1.4	49	4.7	33	4.9	343K	248	504K	101
Q2	7	15	2.0	6.2K	72	18K	167	397K	295	516K	146
Q4	9	26	2.1	32K	26	45K	42	78K	49	113K	25
Q5	7	64	0.5	59	0.6	26	0.7	7.6K	4.5	7.6K	1.5
Q6	7	9	1.4	4.6K	3.9	2.0K	3.1	374K	236	393K	113

5.1 The α Parameter Dominates Heuristic Choice

The most striking finding is that the *search regime*, which is controlled by α in Equation (1), determines performance far more than the choice of heuristic function. Figure 3 visualizes the separation on Q_1: DFS and greedy configurations remain orders of magnitude below BFS and A* across all $\ell_{\min}$ values. Table 1 quantifies this pattern across queries and scale factors.

DFS dominates all configurations on wall-clock time across every query and scale factor, expanding at most 64 states in under 2.1 ms even on SF3. Greedy best-first (α=0) is the only informed configuration that competes: it achieves pruning ratios of 0.82–1.00 with modest state counts, but at 2–10× the wall-clock cost of DFS due to priority queue overhead. A* (α=0.5) is strictly counterproductive for Boolean reachability: it expands tens to hundreds of thousands of states. This makes it comparable or even identical to BFS, while also incurring additional priority queue overhead that often makes it *slower* than BFS in wall-clock time. The $g(s)$ component forces level-by-level expansion that negates the heuristic's directional guidance.

This pattern holds at both scales and strengthens at SF3: as the graph grows, BFS and A* frontier sizes scale with graph density, while DFS state counts remain constant. On Q_1 at $\ell_{\min}$=7, BFS expands 173K states on SF1 versus 504K on SF3 (2.9×), while DFS expands 9 states at both scales. The intermediate configurations WA*-LRS at α=0.3 and α=0.7 interpolate monotonically: α=0.7 consistently matches BFS state counts (pruning = 0.00), while α=0.3 tracks A* closely.

5.2 Comparing Heuristics: Pruning, Slack, and Overhead

Given that greedy best-first is the only viable informed regime, we now compare the three heuristics *within* this regime. Table 2 reports four metrics for each greedy heuristic on SF1. We first clarify what these metrics reveal, then analyze the heuristic comparison.

Reading the metrics. The *pruning ratio* measures the fraction of BFS's search space that a strategy avoids: Prun $= 1 - \text{States}_{\text{greedy}}/\text{States}_{\text{BFS}}$. A pruning ratio of 0.98 means the strategy expanded only 2% of BFS's states. However, a high pruning ratio does not necessarily indicate a strong heuristic since it may reflect the depth-first tendency of greedy search on well-connected graphs.

The *heuristic slack* ($h_{\text{slack}} = g^* - h$, where g^* is the solution depth and h the heuristic estimate at the solution state) measures how far the heuristic's prediction was from reality at the moment a solution was found. A slack of 0 means the heuristic perfectly anticipated the remaining distance; large slack means the heuristic was nearly uninformative about proximity to acceptance.

These two metrics together reveal whether pruning is driven by heuristic quality or by the search strategy's implicit bias.

Example 6 (Strategy Bias). *Consider Q_1 at $\ell_{\min}$=7: Greedy-MAD expands 45 out of BFS's 173,389 states (Prun =1.00), yet h_{slack} = 8.0. At the solution (depth 8), MAD estimated ~0.5 remaining steps. The near-perfect pruning reflects greedy's depth-first tendency on a well-connected graph, not MAD's accuracy: the search dives into one successor, finds a solution quickly, and never materializes the wide frontier that BFS must explore. The heuristic provides tiebreaking among siblings; the depth-first "bias" does the heavy lifting.*

This interpretation is reinforced by Q_4 at $\ell_{\min}$=9, where the pattern partially breaks. Greedy-MAD expands 12,336 out of 51,938 BFS states (Prun = 0.76). This is still substantial pruning, but far from the near-perfect ratios on Q_1 and Q_6. The 4-label cycle (knows/hasInterest/^hasInterest/knows) creates more branching in the product graph, and greedy's depth-first dives hit dead ends more frequently before completing three full cycle iterations. With h_{slack} = 12, MAD's estimate is even less informative here: the heuristic cannot anticipate how many cycle repetitions the graph will require. This is the query where heuristic quality would matter most, but where all three heuristics are weakest.

MAD vs OTD. OTD adds a local topology signal to MAD: among states at the same automaton distance, it prioritizes vertices with more outgoing edges matching the next required NFA transition. When label frequencies are skewed, this signal sharpens the tiebreaking that greedy search relies on.

On Q_1 at $\ell_{\min}$=7, Greedy-OTD expands 25 states versus Greedy-MAD's 45 (a 44% reduction). The `likes/hasCreator` alternation creates vertices with clearly different match ratios: Post vertices have few outgoing `hasCreator` edges (typically one), while Person vertices may have many outgoing `likes` edges. OTD correctly identifies which Person vertices offer more continuation paths and dives through them faster. On Q_6 (`hasCreator/likes`$^+$), the improvement is consistent: 1,389 states versus 2,601.

OTD's one-step lookahead becomes counterproductive when high connectivity does not correlate with progress toward acceptance. On Q_4 at $\ell_{\min}$=9, Greedy-OTD expands 18,084 states versus MAD's 12,336 (47% *increase*). High-degree `knows` hubs produce large match ratios that OTD treats as promising, but these hubs connect to subgraphs that lack the `hasInterest` edges needed to complete the full cycle. OTD aggressively steers the search toward well-connected dead ends that MAD, being topology-agnostic, avoids by chance. A similar pattern appears on Q_2, which also involves `knows` hubs. This confirms that OTD's hypothesis—prioritize vertices with more matching edges—holds when match ratios are discriminative of progress (Q_1, Q_6), but breaks when high connectivity is uncorrelated with path completion (Q_2, Q_4).

Omitted queries. Three of our eight queries are omitted from the tables. Q_7 `(knows/likes)+` and Q_8 `(hasInterest/^hasTag)+` produce no reachable solutions at $\ell_{\min} \geq 5$ on SF1: the product graph is exhausted before paths of the target length exist, so all configurations expand identical state sets with zero pruning. At short path lengths ($\ell_{\min} \leq 2$) where solutions exist, the search space is trivially small (≤30 states) and heuristics provide no meaningful differentiation. Q_3 `(likes/isLocatedIn/^isLocatedIn/^workAt)+` failed to produce valid source vertices across our stratified sampling. We include all eight queries in the experimental setup to be transparent about coverage, but restrict quantitative analysis to the five queries that produce non-degenerate search behavior.

MAD vs LRS. LRS is empirically indistinguishable from MAD on LDBC: states expanded and pruning ratios are identical across all queries (Table 2). The 2-hop mandatory-label check that distinguishes LRS finds no missing labels on this benchmark, because labels like `likes`, `hasCreator`, `knows`, and `hasInterest` appear within 2 hops of virtually every vertex in the LDBC social network. The penalty term evaluates to zero at every expanded state, and LRS reduces to MAD plus the overhead of a wasted bitset lookup.

This overhead is visible in wall-clock time: Greedy-LRS is consistently 1.5–2× slower than Greedy-MAD despite expanding identical state sets (e.g., 4.1 ms vs. 2.1 ms on Q_1 at $\ell_{\min}$=7). On graphs where certain labels are structurally rare the LRS penalty would activate and could provide discrimination that MAD cannot.

Observation 1 (Label Convergence). *To quantify why LRS collapses to MAD, we measured 2-hop label coverage across SF1. The overall median coverage is 75% (6 of 8 labels within 2 hops), and only 11.7% of vertices reach ≥80% coverage. This means the graph is not uniformly label-dense. However, the mandatory labels in our queries are individually near-ubiquitous:* `hasCreator` *and* `hasInterest` *each appear within 2 hops of 99.5% of vertices,* `likes` *reaches 98.9%, and* `knows` *96.3%. LRS's penalty term therefore evaluates to zero at virtually every expanded state. This is not because the graph has uniform label density, but because the labels that recursive social-network queries require happen to be the most pervasive ones. On graphs where mandatory query labels fall in the long tail of the label frequency distribution, the LRS penalty would activate.*

Table 2: Greedy heuristic comparison on SF1 ("all" tier) at representative $\ell_{\min}$. Prun = pruning ratio vs. BFS. h_{slack} = solution depth − h at solution.

Query	ℓ	States	Time (ms)	Prun	h_{slack}
Greedy-MAD					
Q1	7	45	2.1	1.00	8.0
Q2	7	3,897	29	0.98	9.0
Q4	9	12,336	9.3	0.76	12.0
Q5	7	57	0.2	0.91	6.6
Q6	7	2,601	1.8	0.98	8.0
Greedy-OTD					
Q1	7	25	2.0	1.00	8.3
Q2	7	8,558	53	0.95	9.7
Q4	9	18,084	14	0.65	12.0
Q5	7	30	0.3	0.92	6.7
Q6	7	1,389	1.4	0.99	8.5
Greedy-LRS					
Q1	7	45	4.1	1.00	8.0
Q2	7	3,865	50	0.98	9.0
Q4	9	13,680	12	0.74	12.0
Q5	7	57	0.2	0.91	6.6
Q6	7	2,601	2.2	0.98	8.0

Overhead as the binding constraint. Even for the cheapest heuristic, the priority queue imposes measurable overhead relative to DFS. On Q_1 at $\ell_{\min}$=7, DFS expands 9 states in 0.4 ms (∼44 μs/state), while Greedy-MAD expands 45 states in 2.1 ms (∼47 μs/state). The per-state costs are comparable, but Greedy-MAD requires 5× more state expansions to reach a solution that DFS finds immediately via its implicit depth-first bias. For informed search to justify its overhead on Boolean reachability, the heuristic would need to reduce states expanded *below* DFS, a condition not met on the LDBC graphs we used.

5.3 Why Heuristics Are Loose: The Cycle Iteration Problem

The limited discrimination among heuristics traces to a structural cause shared by all recursive RPQs: the NFA's self-loops render minimum-distance estimates nearly uninformative.

Consider `(likes/hasCreator)`$^+$. The NFA has three states with a back-edge from the post-`hasCreator` state to the pre-`likes` state. The minimum automaton distance from any non-accepting state to acceptance is at most 2 transitions, regardless of how many cycle iterations the data graph requires for a given source vertex. MAD therefore returns 1–2 for every expanded state, even when the actual remaining path traverses 6–10 additional edges through multiple

cycle repetitions. Since OTD and LRS are defined as adjustments to the MAD baseline, they inherit this ceiling.

Table 2 quantifies the consequence: h_{slack} (solution depth minus heuristic value at the solution state) ranges from 6.6 to 12 across queries. On Q_1 at ℓ_{min}=7, the best greedy heuristic estimates ~0.5 remaining steps when the true distance is 8. On Q_4 at ℓ_{min}=9, slack reaches 12 for all three heuristics—the 4-state cycle in $(\texttt{knows/hasInterest/\^{}hasInterest/knows})^+$ admits a minimum distance of 4, but solutions require three full iterations.

This is not an artifact of weak heuristic design but a fundamental property of how NFAs represent Kleene-plus recursion: the minimum path through the automaton is agnostic to the number of repetitions the graph will force. Tighter estimates would need to reason about how many cycle completions are reachable from a given vertex, connecting heuristic design to graph-level reachability statistics rather than automaton structure alone.

5.4 Cross-Scale Consistency

All three preceding observations replicate at SF3 (9.0M vertices, 18.0M edges). DFS dominance strengthens at larger scale: its state counts are determined by automaton structure and remain constant across graph sizes (9 states on Q_1 at both SF1 and SF3), while BFS and A* frontier sizes grow with graph density (173K → 504K for BFS on Q1). The heuristic quality metrics (h_{mean}, h_{slack}) are effectively identical across scales, confirming that the loose bounds are a property of NFA cycle structure, not of graph size. The relative ranking of heuristics is: OTD best on expressions with small, skewed label alphabets (e.g. Q_1/Q_6), worst on long sequences with high-degree hub labels (Q_2/Q_4); LRS ≈ MAD everywhere. This observation is stable across scales, and per-state overhead ratios are consistent.

6 Toward Traversal Strategy Selection

Our results establish that informed search, as studied here, does not outperform DFS for Boolean reachability in the in-memory setting. We view this as a necessary empirical baseline rather than a negative result. Several directions emerge from this characterization.

Why DFS wins, and when it might not. DFS dominates because Boolean reachability rewards the first strategy to reach *any* accepting state, and DFS's depth-first exploration finds deep solutions with minimal state materialization. This advantage may diminish under different query objectives. Shortest-path and top-k queries require exploring *all* paths of a given length before certifying optimality; here, A* with an admissible heuristic could prune suboptimal branches that DFS would explore exhaustively. Characterizing these crossover points is the most direct next step.

The α parameter as a design axis. Our unified parameterization (Equation (1)) reveals that the exploration-exploitation tradeoff controlled by α has a first-order effect on both performance and completeness. Greedy search (α=0) achieves depth-first-like behavior with heuristic tiebreaking; A* (α=0.5) collapses to BFS-like behavior with overhead. This suggests that practical systems should treat α as a tunable parameter, which could be potentially selected by a cost model based on query objective, automaton structure, and lightweight graph statistics (e.g. label frequency distribution). Building such a cost model, analogous to join ordering in relational optimization, is a longer-term goal.

Tighter heuristics for recursive queries. The large heuristic slack we observed stems from the NFA's self-loops: MAD returns the minimum automaton distance, which is agnostic to how many cycle iterations the data graph will require. Tighter estimates could incorporate graph-level reachability information: For instance, precomputing the minimum number of cycle completions reachable from each vertex given the local label distribution. Such a heuristic would bridge automaton-only reasoning (MAD) and graph-structural reasoning (OTD, LRS), potentially providing the discrimination that the current heuristics lack on recursive patterns.

Bidirectional search for RPQs. Bidirectional BFS has been studied for RPQs by Koschmieder and Leser [17], who used it as a subroutine within their rare-label decomposition, and by Tetzel et al. [25], who focused on unidirectional traversals and noted the complexity of bidirectional termination for many-to-many RPQ evaluation. Existing work has examined bidirectional search only for specific regular expression families (e.g. transitive restricted expressions [18]) or as a component of composite strategies; a systematic empirical and theoretical study of bidirectional traversal across general recursive RPQs remains missing. We are currently pursuing such a study, including the question of whether heuristic guidance can inform the direction-selection policy.

Weighted edges. Recent papers have looked at path queries that involve using edge and vertex attributes on top of label patterns [13, 14, 19]. Extending to weighted graphs opens the door to heuristics informed by edge-property distributions and enables connections to classical shortest-path search. Weighted A* becomes more natural in this setting, as the $g(s)$ component carries meaningful cost information.

7 Conclusion

We presented the first empirical study of informed search for in-memory RPQ evaluation. Our results show that for Boolean reachability, the search regime parameter α matters more than heuristic sophistication: greedy best-first search achieves substantial pruning over BFS, while A* is counterproductive. DFS remains dominant on wall-clock time, but our analysis reveals the structural reasons (NFA cycle-iteration blindness and label ubiquity) and identifies the conditions under which informed search may prove more valuable for richer query objectives.

Based on our findings, we offer the following preliminary decision rule for practitioners: *use DFS for Boolean reachability when the NFA contains cycles*; *prefer greedy search with OTD when the label alphabet is small and label frequencies are skewed* (as tiebreaking provides measurable pruning); and *reserve A* for shortest-path or top-k objectives*, where its optimality guarantees can offset the priority-queue overhead. Validating the last recommendation is the most immediate next step.

References

[1] [n. d.]. SQL/PGQ. https://duckpgq.org/documentation/sql_pgq/. [Accessed 1-Nov-2025].

[2] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 1–40.

[3] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, and Javiel Rojas-Ledesma. 2022. Time-and space-efficient regular path queries. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 3091–3105.

[4] Jorge Baier, Dietrich Daroch, Juan L. Reutter, and Domagoj Vrgoč. 2017. Evaluating Navigational RDF Queries over the Web. In *Proceedings of the 28th ACM Conference on Hypertext and Social Media* (Prague, Czech Republic). Association for Computing Machinery, New York, NY, USA, 165–174. doi:10.1145/3078714.3078731

[5] Pablo Barceló, Leonid Libkin, Anthony W Lin, and Peter T Wood. 2012. Expressive languages for path queries over graph-structured data. *ACM Transactions on Database Systems (TODS)* 37, 4 (2012), 1–46.

[6] Pablo Barceló Baeza. 2013. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*. 175–188.

[7] Angela Bonifati and Stefania Dumbrava. 2018. Graph queries: From theory to practice. *SIGMOD Record* 47, 4 (2018), 5–16.

[8] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 619–630.

[9] Benjamín Farías, Wim Martens, Carlos Rojas, and Domagoj Vrgoč. 2023. Pathfinder: A unified approach for handling paths in graph query languages. *arXiv preprint arXiv:2306.02194* (2023).

[10] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. 2023. A Researcher's Digest of GQL. In *The 26th International Conference on Database Theory, 2023*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 1–1.

[11] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*. 1433–1445.

[12] Roberto García and Renzo Angles. 2024. Path Querying in Graph Databases: A systematic mapping study. *IEEE Access* (2024).

[13] Amélie Gheerbrant, Leonid Libkin, Liat Peterfreund, and Alexandra Rogova. 2024. Gql and sql/pgq: Theoretical models and expressive power. *arXiv preprint arXiv:2409.01102* (2024).

[14] Amélie Gheerbrant, Leonid Libkin, and Alexandra Rogova. 2025. Dangers of List Processing in Querying Property Graphs. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–25.

[15] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[16] Mahmoud Abo Khamis, Alexandru-Mihai Hurjui, Ahmet Kara, Dan Olteanu, Dan Suciu, and Zilu Tian. 2025. Output-Sensitive Evaluation of Acyclic Conjunctive Regular Path Queries. arXiv:2509.20204 [cs.DB] https://arxiv.org/abs/2509.20204

[17] André Koschmieder and Ulf Leser. 2012. Regular path queries on large graphs. In *Proceedings of the 24th International Conference on Scientific and Statistical Database Management* (Chania, Crete, Greece) *(SSDBM'12)*. Springer-Verlag, Berlin, Heidelberg, 177–194. doi:10.1007/978-3-642-31235-9_12

[18] Qi Liang, Dian Ouyang, Fan Zhang, Jianye Yang, Xuemin Lin, and Zhihong Tian. 2024. Efficient Regular Simple Path Queries under Transitive Restricted Expressions. *Proc. VLDB Endow.* 17, 7 (March 2024), 1710–1722. doi:10.14778/3654621.3654636

[19] Leonid Libkin, Wim Martens, Filip Murlak, Liat Peterfreund, and Domagoj Vrgoč. 2025. Querying Graph Data: Where We Are and Where To Go. In *Companion of the 44th Symposium on Principles of Database Systems*. 9–26.

[20] Alberto O. Mendelzon and Peter T. Wood. 1995. Finding Regular Simple Paths in Graph Databases. *SIAM J. Comput.* 24, 6 (1995), 1235–1258. arXiv:https://doi.org/10.1137/S009753979122370X doi:10.1137/S009753979122370X

[21] Amine Mhedhbi, Amol Deshpande, and Semih Salihoğlu. 2024. Modern Techniques For Querying Graph-structured Databases. *Found. Trends Databases* 14, 2 (Oct. 2024), 72–185. doi:10.1561/1900000090

[22] Maurizio Nolé and Carlo Sartiani. 2016. Regular path queries on massive graphs. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*. 1–12.

[23] Stuart Rusell and Peter Norvig. 2010. Artificial intelligence: A modern approach. (2010).

[24] Michael Sipser. 1996. Introduction to the Theory of Computation. *ACM Sigact News* 27, 1 (1996), 27–29.

[25] Frank Tetzel, Romans Kasperovics, and Wolfgang Lehner. 2019. Graph traversals for regular path queries. In *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 1–8.

[26] Frank Tetzel, Hannes Voigt, Marcus Paradies, Romans Kasperovics, and Wolfgang Lehner. 2018. Analysis of Data Structures Involved in RPQ Evaluation.. In *DATA*. 334–343.

[27] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2015. WAVEGUIDE: Evaluating SPARQL Property Path Queries.. In *EDBT*, Vol. 2015. 525–528.

[28] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2016. Query planning for evaluating SPARQL property paths. In *Proceedings of the 2016 International Conference on Management of Data*. 1875–1889.