Automatic View Selection in Graph Databases

Chao Zhang
University of Helsinki
Finland
Renmin University of China
China

Jiaheng Lu
University of Helsinki
Finland

Qingsong Guo
University of Helsinki
Finland

Xinyong Zhang
Huawei Technologies Co., Ltd.
China

Xiaochun Han
Huawei Technologies Co., Ltd.
China

Minqi Zhou
Huawei Technologies Co., Ltd.
China

ABSTRACT

Recently, several works have studied the problem of view selection in graph databases. However, existing methods cannot fully exploit the graph properties of views, e.g., supergraph views and common subgraph views, which leads to a low view utility and duplicate view content. To address the problem, we propose an extended graph view that persists all the edge-induced subgraphs to answer the subgraph and supergraph queries simultaneously. Furthermore, we present the graph gene algorithm (GGA), which relies on a set of view transformations to reduce the view space and optimize the view benefit. Extensive experiments on real-life and synthetic datasets demonstrated GGA outperformed other selection methods in both effectiveness and efficiency.

CCS CONCEPTS

• **Information systems** → **Database views**;

KEYWORDS

View Selection, Graph Database, Graph Gene Algorithm

1 INTRODUCTION

View selection is a well-studied topic in relational [1, 4, 12, 25], XML [13, 19, 22], and semantic databases [3, 10]. Various methods are proposed to select the materialized views for different target queries, e.g., SQL and XQuery. However, they are not suitable for graph view selection because they do not consider the structural properties of graph queries, e.g., subgraph patterns. Kaskade [7] is a view selection method that inputs the view templates and then generates views as Cypher queries. It modeled the view selection problem as an 0-1 Knapsack problem, and used a branch-and-bound solver to select the graph views. However, there are two major limitations to existing methods.

The first limitation is that existing methods only select views with the subgraph patterns to answer the queries while they do not consider using a view with a supergraph pattern to answer the contained queries. This leads to a low utility of the materialized views. To address this limitation, we propose an *extended graph view*, which is created via an edge-induced method, being capable of answering the subgraph and supergraph queries simultaneously. We also proposed a filtering-and-verification framework to check the query containment by views.

The second limitation is that existing methods cannot effectively explore the possible candidate view combinations to reduce the view space and improve the view benefit. Such a view set $\mathcal{V}$ could be reused to answer other contained queries, thereby saving the view space. However, generating an optimal view set $\mathcal{V}'$ for a query workload Q is challenging due to the exponential search space. In addition, exploring the graph properties among views, e.g., finding the maximum common subgraphs to generate a smaller view set, entails an NP-hard problem of subgraph isomorphism [15]. To mitigate this problem, we propose a graph gene algorithm (GGA), which relies on a three-phase framework that explores graph view transformations to reduce the view space and optimize the benefit.

To summarise, we made the following contributions:

(1) We proposed an *extended graph view*, which is created by an edge-induced method that translates a graph query to a query pattern and persists all its edge-induced subgraphs to answer both subgraph and supergraph queries.
(2) We proposed a filtering-and-verification framework that enables the verification of the query containment by views.
(3) We proposed a view selection algorithm, GGA, to select the views into the memory under a space budget, which explores various options of graph view transformations to find an optimal view set.
(4) We conducted extensive experiments on diverse query workloads and datasets. The results showed that GGA outperformed other selection methods in both effectiveness and efficiency.

2 PRELIMINARIES

This section presents the definitions of terminologies and the view selection problem. Particularly, Section 2.1 defines the property graph, pattern query, edge-induced subgraph, and extended graph view; Section 2.2 defines the view selection problem.

2.1 Graph, Queries and Views

Labeled property graph. A labeled property graph is a multi-relational, attributed, digraph $G = (V_G, E_G, L, P)$, where (1) V_G is a set of vertices; (2) $E_G \subseteq V_G \times V_G$, in which (v, v') denotes an edge from vertex v to v'; (3) L is a label function such that for each vertex $v \in V_G$ (resp. edge $e \in E_G$), L(v) (resp. L(e)) is a label from an alphabet Σ; (4) P is a function such that for each node $v \in V_G$ (resp. edge $e \in E_G$), P(v) (resp. P(e)) is a set of key/value pairs called properties of a node or an edge.

Graph pattern query. A graph pattern query is a digraph $Q_G = (V_p, E_p, L, f)$ over a labeled property graph G, where (1) V_p is a set of query nodes and E_p is a set of query edges, respectively; (3) L is a label function such that for each vertex $v \in V_p$ (resp. edge $e \in E_p$), L(v) (resp. L(e)) is a label from an alphabet Σ; (4) f is a function such that for each vertex $v \in V_p$ (resp. edge $e \in E_p$), f(v) (resp. f(e)) is a Boolean predicate.

Edge-induced subgraph. An edge-induced subgraph is a graph $S = (V_S, E_S, L, P)$ that contains a subset E_S of the edges of a graph G together with any vertices V_S that are their endpoints.

Extended graph view. An extended graph view is a view $V = (V_P, V_G)$, where (1) V_P is a view pattern which is a graph pattern query Q_G with a traversal order of the edges; (2) V_G is the view content that includes all the edge-induced subgraphs S in the traversal order of V_P. Please refer to the extended version [26] of the paper for more details.

2.2 View Selection Problem

Definition 2.1. **(Benefit of a view)**: Given a query workload Q, the benefit b of a view V is the total cost savings by processing the queries using the view V compared to using the graph G:

$$b(V, Q) = \sum_{q \in Q} (w_i \times (cost(q|G) - cost(q|V))) \qquad (1)$$

where w_i is the weight or frequency of query q_i in Q; *cost(q|G)* and *cost(q|V)*, denote the cost of query evaluation over the graph G and view V, respectively. The *cost(q|G)* is calculated depending on the underlying store, e.g., graph store or relational store.

Given a query workload Q and a space budget S, we aim to select an optimal view set $\mathcal{V}$ to materialize within the budget S. Thus, the view selection problem can be modeled as a Knapsack problem of maximizing the view benefit under the space budget.

Definition 2.2. **(View selection problem)**: Given a workload Q and a space budget S, the objective is to select a set of views $\mathcal{V}_s$ derived from a candidate view set $\mathcal{V}$ that fully covers the query results of Q, for maximizing the total benefit of $b(\mathcal{V}_s, Q)$, under the constraint that the total space occupied by $\mathcal{V}_s$ is no greater than S.

The view selection problem is NP-hard [5] for a static single-view case in which $\mathcal{V}_s$ is a subset of $\mathcal{V}$, and each view $V \in \mathcal{V}_s$ is independent so that each query $q \in Q$ is answered by a single view $V \in \mathcal{V}_s$. For such a case, there is a straightforward reduction from the Knapsack problem: find a set of k items with the space occupancy $s_1, \ldots, s_k$ and the benefits $b_1, \ldots, b_k$ so as to maximize the sum of the benefits of the selected items that satisfy the space budget S. Moreover, there could be the dynamic cases in which the views in $\mathcal{V}_s$ can be changed, e.g., by merging, breaking, and removing views. The problem in such cases becomes harder since the space of the candidate view set is extremely huge and it is unfeasible to explore all possible combinations. In addition, for the dynamic case, views are not independent as a query can be answered by multiple views, resulting in a more complicated problem than the static case using the single-view evaluation. In this work, we propose a graph gene algorithm to address the view selection problem in the dynamic multi-view setting.

3 VIEW CONSTRUCTION AND EVALUATION

In this section, we introduce how to construct the candidate view patterns and how to create the view content, as well as how to evaluate the view benefit.

3.1 Edge-Induced View Construction

3.1.1 View pattern construction. Given a candidate query set Q, we translate the queries to a pattern query set, then leverage an edge-induced method to construct a candidate view for each pattern query. Particularly, for a pattern query $Q_G \in Q$, we parse it to Gremlin traversals and derive the traversal patterns E_p; we then add each query edge $e \in E_p$ with the predicates to its view pattern $V_P(Q_G)$ in succession. Since the query node v and edge e are labeled with a given alias, the procedure will also map the alias label to the label $L(v)$ and $L(e)$ in the schema graph.

3.1.2 View content construction. To construct the view content $V_G(Q_G)$, we create an edge-induced graph by the following steps: (i) we traverse each edge $e \in E_p$ in the traversal order as the view pattern $V_P(Q_G)$'s. (ii) for each visited query edge e, we add all the matched results of edges $E(e)$ in the property graph G with their endpoints $V(e)$ to the view content $V_G(Q_G)$. (iii) the procedure terminates when all the patterns have been visited. To the end, the selected graph views are materialized in the format of GraphML [24], which is an XML-based representation of a graph.

3.2 A Filtering-and-Verification Framework

The filtering-and-verification framework consists of two stages. The first stage will check if a pattern query Q_G is contained by a view pattern $V_P(Q'_G)$. Otherwise, the query will not be evaluated on view V. The second stage will further verify if the view content $V_G(Q_G)$ contains all the matched results of the given query. Intuitively, the first stage checks the containment between a query pattern and a view pattern, and the second stage verifies the containment between query results and the view content.

3.2.1 The filtering stage. In this stage, we check if a pattern query Q_G is a subgraph pattern of a view pattern $V_P(Q'_G)$. It is known that finding all the subgraph isomorphism mappings is NP-hard [15], but there exist several practical algorithms to decide the answers in polynomial time. In this work, we employ the VF2 algorithm [6] to the pattern containment between two graph patterns.

3.2.2 The verification stage. This stage verifies if a query Q_G can be answered by the view content $V_G(Q'_G)$. It checks if the following conditions hold: (i) there exists a mapping M from each edge pattern $e \in E_p$ to the edge pattern $e' \in E'_p$. (ii) for the edge $e' \in E'_p$ that has no mapping from $e \in E_p$, if the vertex $v'_e \in e'$ has the mapping from $v_e \in e$, the node $v'_e = M(v)$ must have occurred in prefix traversal patterns of E'_p.

3.2.3 The evaluation of view benefit. Once the framework has verified the containment of a pattern query Q_G and a view V, G-View then evaluates the benefit $b(Q_G, V)$. In our implementation, we use the PROFILE feature [24] of Gremlin to obtain the cost of query evaluation. Specifically, the PROFILE step returns various metrics about the given Gremlin queries including the result size, count of traversals, and total execution time in each pipeline. We perform the PROFILE step over the view V and over the G, respectively. Then we take the total execution time as the cost and compute the benefit according to Equation 1.

4 GRAPH GENE ALGORITHM

In this section, we propose the graph gene algorithm (GGA) for view selection. Specifically, Section 4.1 introduces the view transformations. Section 4.2 presents the GGA algorithm.

4.1 View Transformations

The GGA algorithm is inspired by the gene algorithm (GA) [2], it encodes the view patterns as graph genes and solves the view selection problem as a state search process. By merging, breaking, and removing views from the initial state, we obtain another state from view set $\mathcal{V}'$ with a new benefit $b(\mathcal{V}')$. Particularly, GGA has three atomic behaviors for view pattern transformations, namely, FISSION, FUSION, and REMOVE. In the following, we introduce the view transformations in detail.

4.1.1 FISSION transformation. This transformation splits a view pattern to multiple genes. Specifically, we find the articulation points of a view pattern by using the Tarjan Algorithm [23], then obtain multiple graph genes by breaking down the view pattern according to its articulation points. The articulation points are vertices whose removal increases the number of connected components of the graph, and Tarjan Algorithm [23] is a (Depth-First-Search) DFS-based approach that can run in *O(V+E)* time to compute the articulation points in a directed graph. If the articulation point does not exist, the view pattern becomes the graph gene itself.

4.1.2 FUSION transformation. FUSION is opposite to FISSION, namely, this transformation merges or joins a view $V_i \in \mathcal{V}$ to another view $V_{j \neq i} \in \mathcal{V}$. Particularly, FUSION has two variants:

(1) Merge a sub-view $V_i \subset V_j$: Fusion merges the view V_i to $V_{j \neq i} \in \mathcal{V}$ if V_i is contained by $V_{j \neq i}$. It requires (1) V_i is a subgraph of $V_{j \neq i}$; (2) V_i has a prefix traversal pattern of $V_{j \neq i}$'s.

(2) Merge-join the genes $g_i \subset g_j$: Fusion merges the genes $g_i \in V_i$ if $g_j \in V_{j \neq i}$ contains g_i; the remaining genes $g_{k \neq i} \in V_i$ are joined to $V_{j \neq i}$ if they are not contained by other views $V_{k \neq i,j}$.

The first case can be decided via the filtering-and-verification framework, and contained views can be merged directly. For the second case, the algorithm enumerates all the genes $g_i \in V_i$ over the view set $\mathcal{V}$ to check the containment on other graph genes $g_j \in V_{j \neq i}$ via the filtering-and-verification framework, then merges them to the contained genes if any. The remaining genes $g_{k \neq i} \in V_i$ are assembled to the view $V_{j \neq i}$ that has contained genes by connecting the articulation points.

4.1.3 REMOVE transformation. REMOVE eliminates the empty-gene candidate views after a sequence of view transformations. Such candidate views can be removed as they have been contained by other views.

EXAMPLE 1. *Figure 1 illustrates the view transformations. Given a view pattern $V_P(Q_G)$ and a view set $\mathcal{V} = \{V_1, V_2, V_3\}$, GGA applies a set of transformations on the view patterns. In the FISSION phase, $V_P(Q_G)$ is broken down to four genes based on the articulation points $\{B, C, E\}$. Then the genes are merged to the view set in the FUSION phase. Specifically, genes 1,2,3 are merged to V_1, V_2, V_3, respectively, and the remaining gene 4 is joined to V_3 on node E. Finally, the $V_P(Q_G)$ is removed from the candidate view set and we have reduced the common parts of three graph genes of it, i.e., genes 1, 2, and 3.*

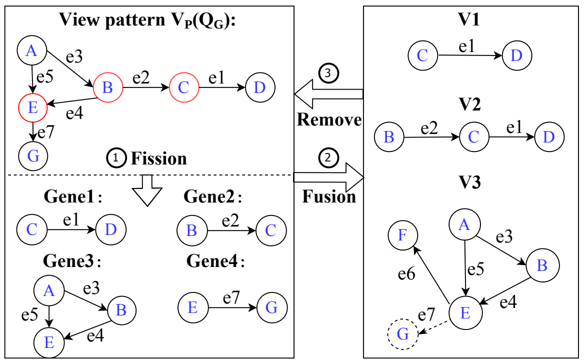


Figure 1: An illustration of view transformations.

Algorithm 1: Graph-Gene Algorithm (GGA)

```
Input: A query workload $Q$, a candidate view set $\mathcal{V}$, a space budge
       $S$, fission probability $p_f$, fusion probability $p_c$
Output: A view set $\mathcal{V}_s$.
1  while !timeout do
2      for $i \leftarrow 1$ to $\mathcal{V}.length$ do
3          if random(0,1) $< p_f$ then
4              $\mathcal{V} = \mathcal{V}_{index \neq i} \cup FISSION(\mathcal{V}_i)$        // Fission
5          if random(0,1) $< p_c$ then
6              $\mathcal{V} = FUSION(\mathcal{V}_{index \neq i}, \mathcal{V}_i)$           // Fusion
7          if $\mathcal{V}_i.genes = empty$ then
8              $\mathcal{V} = REMOVE(\mathcal{V}, \mathcal{V}_i)$                // Remove
9      $B_v = SearchAndEvaluate(\mathcal{V}, Q)$ // Evaluation
10     $\mathcal{V}' = DPS(\mathcal{V}, B_v, S)$ // View selection
11     if $B'_V > B_V$ then
12         $\mathcal{V}_s = \mathcal{V}'$
13     else
14         skip $\mathcal{V}'$
15 return $\mathcal{V}_s$
```

4.2 GGA Algorithm Description

Integrating the methods of view transformations and benefit evaluation, we devise a view selection algorithm, called the GGA (Graph-Gene Algorithm), which is shown in Algorithm 1. Given a query workload Q and a candidate view set $\mathcal{V}$, it returns a view set $\mathcal{V}_s$ that is transformed from $\mathcal{V}$. In addition, two probabilities p_f and p_f are provided to perform the random FISSION and FUSION transformations, respectively.

When the termination condition, e.g., a timeout threshold or an iteration number, is not satisfied, GGA repeatedly applies the FISSION, FUSION and, REMOVE transformations to derive a new state of the view selection (lines 1-8). To simplify the description, we assume the selection procedure is conducted by the helper procedure *SearchAndEvaluate* (line 9), which calls the methods of benefit evaluation for multiple views [26]. We use the cost optimizer [24] of Gremlin to evaluate the view benefit and In line 9. The algorithm calls another helper function *DPS* to select the views based on the dynamic programming strategy.

The function of dynamic programming selection (*DPS*) goes as follows: (i) initialize a benefit vector B_V and a size vector S_V. We leverage the PROFILE [24] of Gremlin to derive the size vector S_V, one can also plug other size estimators, e.g., [11], to obtain it; (ii) fill the DP table by considering two cases for every view: (a) the view is included in the optimal subset, (b) not included in the optimal set. Therefore, the maximum value that can be obtained according to the equation: DP[i][j] = max(B_V[i] + DP[i-1][j-S_V[i]],DP[i-1][j]). (iii) use a bottom-up approach to obtain the optimal selection $\mathcal{V}'$.

Note that the algorithm only jumps to a new state with a higher benefit. Otherwise, it will skip the current state and continue applying transformations to the views that are from the previously obtained state to reach a another state (lines 11-14). When the termination condition is satisfied, the algorithm returns an optimal view selection under the space budget.

Definition 4.1. **(Transformation Completeness)**: Let $\mathcal{V}$ be a set of candidate views and $\mathcal{V}^i$ be the i-th state of the candidate view set. $\mathcal{V}^i$ is transformation complete iff there exists a set of sequence transformation T = $\{\tau_1, \tau_2, \ldots, \tau_n\}$ such that $\mathcal{V}$ and $\mathcal{V}^i$ cover the same workload Q.

Lemma 1. *Any state of view sets in GGA algorithm is transformation complete for a candidate view set* $\mathcal{V}$.

Proof 1. *(Sketch) The transformation set T= {FISSION, FUSION, REMOVE} is complete for any candidate view set* $\mathcal{V}$*. Firstly, FISSION breaks the initial view set* $\mathcal{V}$ *to a fine-grained view set with graph genes. Thus, the joined view content of these graph genes can cover the view content of* $\mathcal{V}$*. Secondly, FUSION merges the view set* $\mathcal{V}'$ *with overlap genes. Hence, the union of view content of the remaining genes still covers the view content of* $\mathcal{V}$*. Finally, the empty-gene views are eliminated by REMOVE but they can be answered by other views. Therefore, for any state of view set, the original workload Q can be covered by a new view set* $\mathcal{V}$*. That concludes the proof.*

5 EXPERIMENTS

Experimental Setup. All the experiments were conducted on a machine with a 2-core i5 CPU (2.9 GHz) and 16GB RAM. We implemented all the compared methods in JAVA 1.8. We deployed SQLG v2.0.2 to stored the raw data. We constructed the views from SQLG and materialized them to GraphML [24] files.

Datasets and Workloads. We used both synthetic and real-life data to conduct the experiments. We used a synthetic social network dataset from the LDBC benchmark [8], which includes 11 entities connected by 20 relations. We generated an LDBC graph with the scale factor SF1, resulting in a graph with roughly 1M vertices and 2M edges. We designed a workload including 12 pattern queries following [14]. We used two real-life graphs: (a) Amazon dataset [16], a product co-purchasing network with 542K nodes and 3.3M edges. We designed 12 frequent query patterns following [17], where each of the view content contains 67K nodes and edges on average. (b) DBLP-citation network [21], a bibliography that contains the publication information and co-authorship in computer science. The dataset has 1M nodes and 2M edges. We also identified 12 query patterns similar to the Amazon patterns. Details of the query patterns can be found at the extended version [26].

Compared Selection Algorithms. We measured the performance of the view selection algorithms in the experiments. Specifically, all the algorithms modeled the selection problem as a Knapsack problem and they aimed at selecting the extended graph views under a space budget for a given workload. We conducted three sets of experiments to evaluate (1) the effectiveness of the algorithms in answering the query, reducing the view size, and optimizing the view benefit; (2) the efficiency of the selection algorithms; and (3) the convergence of the GGA algorithm. We compared the following selection algorithms with GGA:

(1) **Dynamic Programming Selection (DPS):** Our first baseline method is the selection method based on dynamic programming, which is described as a function in Section 4.2.

(2) **Greedy-Based Selection (Greedy):** The second algorithm is a greedy-based algorithm [22]. In particular, this method computes the view benefit in each iteration and remove a view with the maximum benefit, along with the queries it contained. The algorithm terminates until all queries are included or the total size exceeds the size constraint.

(3) **Kaskade**: The third algorithm is a method used by Kaskade [7]. We implemented it as follows: (i) we input the view templates with no containment to simulate its view enumeration; (ii) we enumerate the queries and evaluate the benefit of a view that contains the current query to simulate its single-view rewriting; (iii) we leverage the PROFILE [24] step to derive the size vector; and finally (iv) we use a branch-and-bound solver to select the views.

5.1 Effectiveness of Selection Algorithms

We ran four selection algorithms. Namely, DPS, Greedy, Kascade, and GGA, to evaluate their effectiveness. We tested the algorithms by varying the space budgets with S/6, S/4, and S/2, where S denotes the total view size $\sum_v s(v)$.

Figure 2 depicts the performance of selection algorithms in optimizing the view benefit. Overall, for any workload and space budget, the GGA algorithm achieved the highest view benefit, thus can have the largest query processing cost reductions. For the LDBC dataset with S/2, it improved 36%, 20%, 19% of view benefit over DPS, Greedy, and Kascade, respectively. For the Amazon dataset with S/2, it achieved 30%, 20%, 9% of view benefit improvement, respectively. The view benefit was significantly improved by GGA algorithm by 70%, 150%, 53% in the DBLP dataset. Kascade had a higher benefit than DPS and Greedy because (i) it has eliminated the contained views, thus it selected more useful views than DPS, and (ii) it used the branch-and-bound strategy to search the solution, thus can optimize both view benefit and space. Nevertheless, it has an averagely 27% lower benefit than GGA. This is mainly attributed to (1) GGA's fine-grained view transformations that explore and merge the views with common subgraph parts. (2) its benefit evaluation strategy that can take multiple view combinations to optimize the benefit.

Figure 3 illustrates the fraction of queries that can be answered by the selected views. GGA clearly outperformed others because it employed supergraph views, merged views, and view combinations, which result in more contained queries. Particularly, it can fully cover all the queries when the space budget is increased to S/2. DPS had the lowest query coverage in the LDBC and Amazon datasets because it selected the views independently. Greedy had a higher query coverage than DPS because it removed the contained queries in each round. However, the query fraction of Greedy was affected by the low-utility views that have a high benefit and a large size in

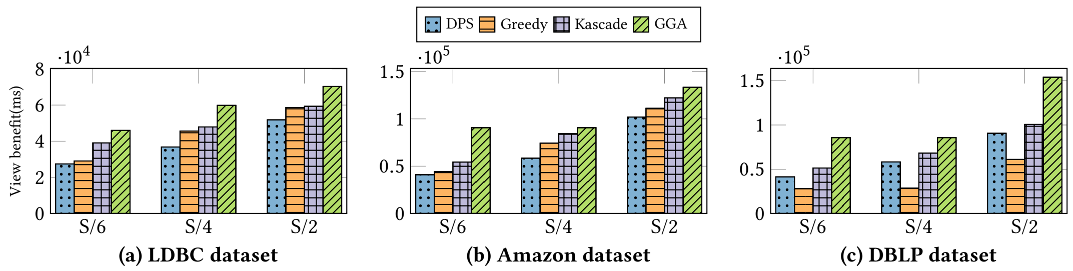


Figure 2: View benefit for the workloads in three datasets based on views selected by three algorithms.

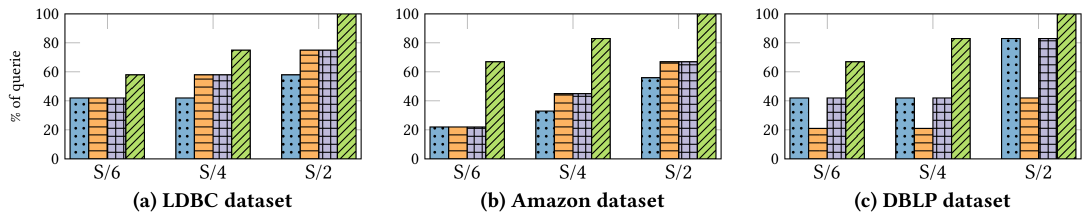


Figure 3: Fraction of queries for the workloads in three datasets covered by views selected by three algorithms.

the DBLP dataset. Kascade can address this issue with its branch-and-bound solver, but still, it can not compete with GGA because it only considers single-view query rewriting.

Figure 4a illustrates the size of candidate views generated by the selection algorithms. DPS had the largest view size because it had a candidate view for each query, while Greedy and Kascade had the same and relatively smaller size because they pruned the contained views. It is clearly visible that GGA method outperformed others because of its gene-based view transformation and combination. Particularly, it reduced the space of the view size by up to 61%, 60%, 58% for LDBC, Amazon, and DBLP, respectively.

5.2 Efficiency of Selection Algorithms

Figure 4b shows the running time of four algorithms in milliseconds. In particular, the time consists of the execution time for view construction, view evaluation, and view selection. The results manifested that GGA outperformed others regarding efficiency. Overall, it accelerated 36%, 20%, 30% of running time of DPS, Greedy, and Kascade for the LDBC workloads, respectively. The improvement was achieved up to 58%, 71%, and 55% for the Amazon workloads, and 19%, 59% and 16% for the DBLP workloads. Kascade was faster than DPS because it had a reduced candidate set after the view enumeration. Greedy incurred significant overhead because it had to re-evaluate the view benefit in each round. The primary advantage of GGA over others is that it has reduced the number and size of views in the candidate set, thus saved unnecessary computation of view evaluation. For the view selection phase, GGA was the best because it had the smallest candidate set to select and generate.

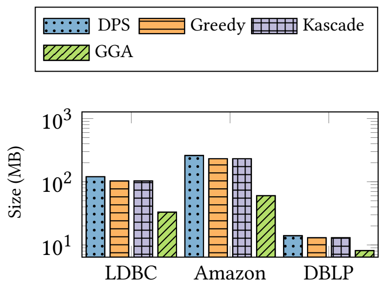
5.3 The Convergence of GGA

In this experiment, we investigated the convergence of GGA. We set both fission and fusion probabilities to 50% and ran the algorithm with space budget S/2. The result was shown in Figure 4c, which confirmed that GGA is effective: the algorithm converges within 10 generations for the workloads in three datasets. Furthermore, the results indicated that the strategy of state search is effective. When a state of view selection has a lower benefit than the previous state, the algorithm can jump to another state with a higher benefit.

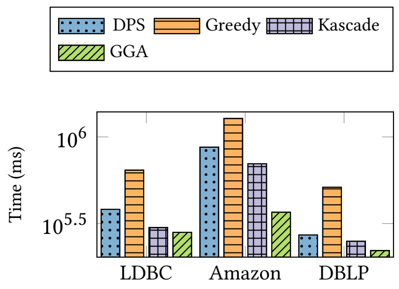
6 RELATED WORK

View selection for relational, XML and RDF data. Materialized view selection in relational databases has been a well-studied topic (see [5, 18] for surveys). Particularly, Chaves et al. [4] encoded the relational views as genes and applied the gene algorithm to the view selection problem in the setting of distributed databases. Recently, there emerged work, e.g., [25], that utilized deep reinforcement learning to guide the view selection. There has been a host of work on processing XML queries using views [13, 19, 22]. In [22], the authors studied the view selection problem for XPath workloads, they proposed a greedy-based solution that makes the space/time tradeoff. Katsifodimos et al. [13] studied the view selection for XQuery workloads. They first developed a greedy-based algorithm for a Knapsack selection problem, then proposed a heuristic algorithm to search for an optimal view set based on multi-view rewriting. There has also been work for RDF view selection [3, 10]. Goasdoué et al. [10] solved the view selection problem as a search process. They proposed heuristic strategies to search for a set of reformulated RDF views to minimize the defined cost model. Unfortunately, none of these works considered the structural properties of graph queries in view selection, thus they cannot be applied directly to the graph view selection problem.

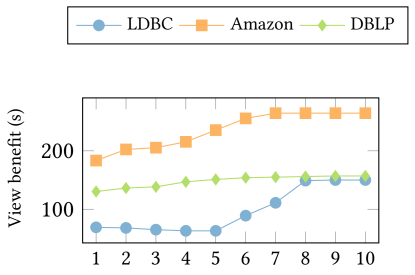
View selection in graph databases. Regarding view selection in graph databases, Fan et al. [9] studied the minimal and minimum containment problems but they considered the views were precomputed and static, leading to duplicate view content. Kascade [7] considered the view selection problem as an 0-1 Knapsack problem,



(a) Candidate view size



(b) Running time



(c) Convergence of GGA

Figure 4: View size, running time and converge of selection algorithms.

which generated the candidates using constraint-based view enumeration, and it used a branch-and-bound solver to select the views. Our work modeled the selection problem as an 0-1 Knasack problem as well. While Kascade only supported single-view rewriting, our GGA algorithm considered the subgraph/supergraph views, view transformations, and multi-view combinations, yielding a view set with a smaller view size and a higher view benefit.

7 CONCLUSION

In this work, we proposed an *extended graph view*, which can answer both the subgraph and supergraph queries. We devised a filtering-and-verification framework to check the query containment by views. We developed a search-based algorithm, GGA, which explores graph view transformations to reduce the view size and optimize the overall query performance. The experimental results manifested that GGA outperformed other selection methods concerning effectiveness and efficiency. In the future, we plan to extend our techniques to other graph query languages such as Cypher [20].

ACKNOWLEDGMENTS

This work is supported by Academy of Finland (310321), China Scholarship Council fellowship (201606360137), and Huawei Research Project (YBN2018125169).

REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R Narasayya. 2000. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, Vol. 2000. 496–505.

[2] David Beasley, David R Bull, and Ralph Robert Martin. 1993. An overview of genetic algorithms: Part 1, fundamentals. *University computing* 15, 2 (1993), 56–69.

[3] Roger Castillo and Ulf Leser. 2010. Selecting materialized views for RDF data. In *International Conference on Web Engineering*. Springer, 126–137.

[4] Leonardo Weiss F Chaves, Erik Buchmann, Fabian Hueske, and Klemens Böhm. 2009. Towards materialized view selection for distributed databases. In *EDBT*. 1088–1099.

[5] Rada Chirkova, Jun Yang, et al. 2012. Materialized views. *Foundations and Trends® in Databases* 4, 4 (2012), 295–405.

[6] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *TPAMI* 26, 10 (2004), 1367–1372.

[7] Joana MF da Trindade, Konstantinos Karanasos, Carlo Curino, Samuel Madden, and Julian Shun. 2020. Kaskade: Graph Views for Efficient Graph Analytics. In *ICDE*.

[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 *SIGMOD*. ACM, 619–630.

[9] Wenfei Fan, Xin Wang, and Yinghui Wu. 2014. Answering graph pattern queries using views. In *ICDE*. IEEE, 184–195.

[10] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. 2011. View Selection in Semantic Web Databases. *PVLDB* 5, 2 (2011), 97–108.

[11] Andrey Gubichev. 2015. *Query Processing and Optimization in Graph Databases*. Ph.D. Dissertation. Technische Universität München.

[12] Himanshu Gupta and Inderpal Singh Mumick. 2005. Selection of views to materialize in a data warehouse. *IEEE Transactions on Knowledge and Data Engineering* 17, 1 (2005), 24–43.

[13] Asterios Katsifodimos, Ioana Manolescu, and Vasilis Vassalos. 2012. Materialized view selection for XQuery workloads. In *SIGMOD*. 565–576.

[14] LDBC task force. 2019. *The LDBC social network benchmark (version 0.3.2)*. Technical Report. Linked Data Benchmark Council.

[15] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proceedings of the VLDB Endowment* 6, 2 (2012), 133–144.

[16] Jure Leskovec, Lada A Adamic, and Bernardo A Huberman. 2007. The dynamics of viral marketing. *TWEB* 1, 1 (2007), 5–es.

[17] Jure Leskovec, Ajit Singh, and Jon Kleinberg. 2006. Patterns of influence in a recommendation network. In *PAKDD*. Springer, 380–389.

[18] Imene Mami and Zohra Bellahsene. 2012. A survey of view selection methods. *Acm Sigmod Record* 41, 1 (2012), 20–29.

[19] Bhushan Mandhani and Dan Suciu. 2005. Query caching and view selection for XML databases. In *VLDB*. VLDB Endowment, 469–480.

[20] Neo4j. 2021. Cypher: the Neo4j graph query Language. https://neo4j.com/cypher-graph-query-language/.

[21] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. 2008. Arnetminer: extraction and mining of academic social networks. In *SIGKDD*. 990–998.

[22] Nan Tang, Jeffrey Xu Yu, Hao Tang, M Tamer Özsu, and Peter Boncz. 2009. Materialized view selection in XML databases. In *DASFAA*. 616–630.

[23] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

[24] Apache Tinkerpop. 2020. https://tinkerpop.apache.org/docs/3.4.4/.

[25] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. 2020. Automatic View Generation with Deep Learning and Reinforcement Learning. ICDE.

[26] Chao Zhang, Jiaheng Lu, Qingsong Guo, Xinyong Zhang, Xiaochun Han, and Minqi Zhou. 2021. Automatic View Selection in Graph Databases. https://arxiv.org/abs/2105.09160.