Learning to Speed Up Query Planning in Graph Databases

AAAI Press
Association for the Advancement of Artificial Intelligence
2275 East Bayshore Road, Suite 160
Palo Alto, California 94303

Abstract

Querying graph structured data is a fundamental operation that enables important applications including knowledge graph search, social network analysis, and cyber-network security. However, the growing size of real-world data graphs poses severe challenges for graph databases to meet the response-time requirements of the applications. Planning the computational steps of query processing – *Query Planning* – is central to address these challenges. In this paper, we study the problem of learning to speedup query planning in graph databases towards the goal of improving the computational-efficiency of query processing via training queries. We present a *Learning to Plan* (L2P) framework that is applicable to a large class of query reasoners that follow the Threshold Algorithm (TA) approach. First, we define a generic search space over candidate query plans, and identify target search trajectories (query plans) corresponding to the training queries by performing an expensive search. Subsequently, we learn greedy search control knowledge to imitate the search behavior of those target search trajectories. We provide a concrete instantiation of our L2P framework for STAR, a state-of-the-art graph query reasoner. Our experiments on several benchmark knowledge graphs including DBpedia, YAGO, and Freebase show that using the query plans generated by the learned search control knowledge, we can significantly improve the speed of STAR with little or no loss in accuracy.

Introduction

Database technology has been successfully leveraged to improve the scalability and efficiency of artificial intelligence (AI) and machine learning (ML) algorithms (Niu et al. 2011; Sarkhel et al. 2016; Das et al. 2016; Zhang, Kumar, and Ré 2016). This paper focuses on opposite direction of this successful cross-fertilization. We investigate ways to improve the computational-efficiency of querying databases by leveraging the advances from AI search, planning, and learning techniques. In this work, we study the following problem: *how can we automatically improve the speed of generating high-quality query plans, for minimizing the response time to find correct answers, by analyzing training queries drawn from a target distribution?*

Graph representations are employed for modeling data in many real-world applications including cyber security, knowledge graphs, social and biological networks. Graph querying is the most primitive operation for information access, retrieval, and analytics over graph data that enables applications including knowledge graph search, and cyber-network security. We consider the problem of querying a graph database, where the input is a *data graph* and a *graph query*, and the goal is to find the answers to the given query by searching the data graph. For example, to detect potential threats, a network security expert may want to "find communication patterns matching the attack pattern of a newly discovered Worm" over a cyber network (Chin Jr et al. 2014). This natural language query is issued to a graph database using a formal graph query language like SPARQL. Specifically, we study the general top-k graph querying problem as illustrated in Example 1.

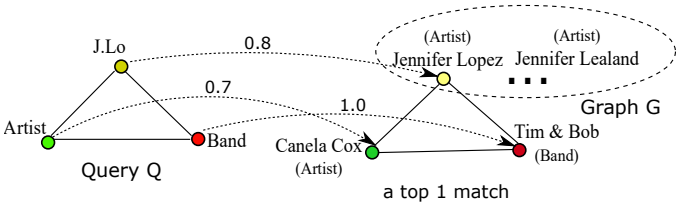

Figure 1: Illustration of Top-k graph querying problem.

Example 1: Consider a graph query Q on DBpedia knowledge graph, shown in Figure 1. We want to find the artists who work with "J.Lo" in a Band. Each of these ambiguous nodes in Q can have excessive number of candidate matches. For example, there are 31,771 nodes with label "Band" and 82,853 "Artists" in DBpedia. In addition, "J.Lo" can match to different people whose first name starts with "J" or last-name starts with "Lo". While "Jennifer Lopez" is the perfect match for this query, users may want to get other similar people like "Jennifer Hudson", etc. However, the matching score for each candidate node can be different due to different node context. It is computationally expensive to first expand all of those matches to find the query structure and then rank them based on their matching scores. □

The growing size of real-world data graphs poses severe challenges for graph databases to meet the response-time requirements of the applications. For example, knowledge

graphs employed in health-care, and cyber networks employed in network monitoring applications may involve billions of linked entities with rich relational information. Planning the computational steps of query processing – *Query Planning* – is central to address these challenges.

In this work, we answer the main research question in the context of Threshold Algorithm (TA) framework (Fagin, Lotem, and Naor 2003) for query processing that is widely popular in both graph and relational databases. The TA framework works as follows. First, the given query is decomposed into sub-queries (e.g., star shaped queries for graph query). Subsequently, it follows a fixed *fetch-and-verify* strategy iteratively until a termination criterion (via estimates of lower and upper bound) is met. Nevertheless, the framework has two major drawbacks: 1) It is very conservative in the upper bound estimation. This often leads to more computation without improvement in results; and 2) It applies static fetch and join policy for all the queries that will likely degrade its performance over heterogeneous query set. In their seminal Gödel prize-winning work, Fagin and colleagues suggested the usefulness of finding good heuristics to further improve the computational-efficiency of TA framework as an interesting open problem (Fagin, Lotem, and Naor 2003). Our work addresses this open problem with strong positive results. *To the best of our knowledge, this is the first work that tightly integrates learning and search to improve the computational-efficiency of query processing over graph databases*.

We develop a general *Learning to Plan* (L2P) framework that is applicable to a large class of query reasoners that follow the Threshold Algorithm (TA) approach. First, we define a generic search space over candidate query plans, where the query plan of TA framework can be seen as a greedy search trajectory. Second, we perform a computationally-expensive search (heuristic guided beam search) in this search space, to identify target search trajectories (query plans) corresponding to the training queries, that significantly improve over the computation-time of query plans from the TA framework. Third, we learn greedy policies in the framework of *imitation learning* to mimic the search behavior of these target trajectories to quickly generate high-quality query plans. We also provide a concrete instantiation of our L2P framework for STAR, a state-of-the-art graph query reasoner and perform a comprehensive empirical evaluation of L2P-STAR on three large real-world knowledge graphs. Our results show that L2P-STAR can significantly improve the computational-efficiency of query processing over STAR with negligible loss in accuracy.

Outline of the Paper. The remainder of the paper is organized as follows. First, we provide background on graph query processing and TA framework that forms the basis of this work. Second, we give our problem setup for learning greedy control knowledge to produce high-quality query plans. Next, we present our learning to plan (L2P) framework, followed by a concrete instantiation of L2P for STAR query processing. We finally present experimental results and conclude.

Background

In this section, we provide the background on graph query processing and the general TA framework as applicable to both graph and relational databases.

Data Graph. We consider a labeled and directed data graph G=$(V, E, \mathcal{L})$, with node set V and edge set E. Each node $v \in V$ (edge $e \in E$) has a label $\mathcal{L}(v)$ ($\mathcal{L}(e)$) that specifies node (edge) information, and each edge represents a relationship between two nodes. In practice, $\mathcal{L}$ may specify heterogeneous attributes, entities, and relations (Lu et al. 2013).

Graph Query. We consider query Q as a graph (V_Q, E_Q). Each *query node* in Q provides information/constraints about an entity, and an edge between two nodes specifies the relationship posed on the two nodes. Formal graph query languages including SPARQL (Prud'Hommeaux, Seaborne, and others 2008), Cypher (Corporation 2013), and Gremlin (O'Brien et al. 2010) can be used to issue graph queries to a database such as Neo4j(`neo4j.com`). Since existing graph query languages are essentially subgraph queries, we can invoke a query transformer to work with different query languages (Kim et al. 2015). Therefore, our work is general and is not tied to any specific query language.

Subgraph Matching. Given a graph query Q and a data graph G, a match of Q in G is a mapping $\phi \subseteq V_Q \times V$, such that (1) ϕ is an injective function, *i.e.,* for any pair of distinct nodes u_i and u_j in V_Q, $\phi(u_i) \neq \phi(u_j)$; and (2) for any edge $(v_i, v_j) \in E_Q$, $(\phi(v_i), \phi(v_j)) \in E$. The match $\phi(Q)$ is a *complete* match if $|Q| = |\phi(Q)|$, where $|Q|$ denotes the size of Q, *i.e.,* the sum of the number of nodes and edges in Q (similarly for $|\phi(Q)|$). Otherwise, $\phi(Q)$ is a *partial match*.

Matching Score. Given Q and a match $\phi(Q)$ in G, we assume the existence of a scoring function $F(\cdot)$ which computes, for each node $v \in V_Q$ (resp. each edge $e \in E_Q$), a matching score $F(\phi(v))$ (resp. $F(\phi(e))$). The matching score of ϕ is computed by a function $F(\phi(Q))$ as

$$F(\phi(Q)) = \sum_{v \in V_Q} F(v, \phi(v)) + \sum_{e \in E_Q} F(e, \phi(e)) \qquad (1)$$

One can use any similarity function such as acronym, abbreviation, edit distance etc. We adopt Leveneshtein function (Navarro 2001) to compute node/edge label similarity, and employ the R-WAG's ranking function that incorporates both label and structural similarities (Roy, Eliassi-Rad, and Papadimitriou 2015) to compute matching score $F(\cdot)$.

Top-k Graph Querying. Given Q, G, and $F(\cdot)$, the top-k subgraph querying problem is to find a set of k matches $Q(G, k)$, such that for any match $\phi(Q) \notin Q(G, k)$, for all $\phi'(Q) \in Q(G, k)$, $F(\phi'(Q)) \geq F(\phi(Q))$.

Top-k Search Paradigm. Top-k graph querying problem is known to be intractable (NP-hard). The common practice in existing top-k graph search techniques is to follow a conventional Threshold Algorithm (TA) style approach (Ding et al. 2014; Zeng et al. 2012; Zou et al. 2014; Yang et al. 2016).

TA **Framework for Query Processing.** The TA framework was originally developed for relational databases (Fagin,

Algorithm 1 TA Framework for Graph Search

Input: a graph query Q, a data graph G, integer k
Output: top-k match set $Q(G, k)$

Decompose Q into a set of sub-queries $\mathcal{Q}$
repeat
  fetch k partial matches for sub-queries in round-robin way
  join to assemble new matches;
  update lowerbound (LB) and upperbound (UB);
until UB $>$ LB
return $Q(G, k)$

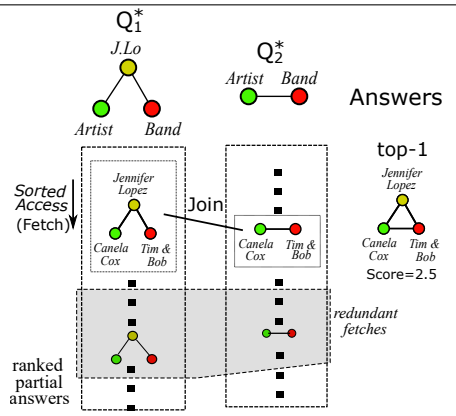

Figure 2: Illustration of TA framework via STAR query processing.

Lotem, and Naor 2003), and has been extended to graph databases as well. It consists of three main steps.

Query Decomposition. A query Q is decomposed into a set of sub-queries $\{Q_1, \ldots, Q_T\}$ without loss of generality. The procedure then initializes one list for each sub-query to store the partial answers. For relational databases, sub-queries can correspond to individual attributes (Fagin, Lotem, and Naor 2003); and for graph databases, sub-queries can correspond to nodes or edges or stars (Yang et al. 2016) or spanning trees (Cheng, Zeng, and Yu 2013).

Partial Answer Generation. For each sub-query Q_i, it performs a iterative exploration as follows. 1) It computes and fetches k partial matches of Q_i; and 2) It verifies if the partial matches can be joined with other "seen" partial matches to be able to improve the top-k matching score. The above step is repeated until k matches are identified and it is impossible to generate better matches.

Early Termination. TA dynamically maintains 1) a lower bound (LB) as the smallest top-k match score so far; and 2) an upper bound (UB) to estimate the largest possible score of a complete match from unseen matches. For example, an upper bound can be established by aggregating the score of the next match from each list. If UB is smaller than the current lower bound LB, TA terminates.

Figure 2 illustrates TA framework for the query Q shown in Figure 1, where each sub-query takes the form of a star (i.e., graph with a unique pivot and some adjacent nodes).

Limitations. The main limitations of TA framework are as follows. 1) Applying fixed fetch-and-verify strategy for all the queries (*e.g.,* always fetch a fixed amount of partial answers) may not provide robust performance due to diversity in queries; 2) The performance of TA framework highly depends on a good estimation of the upper bound. Enforcing a fixed estimation scheme can be loose, leading to "redundant fetches" with little improvement in quality of answers (see Figure 2). More adaptive stopping criterion is desirable; and 3) It is hard to adapt TA framework to different *resource constraints* (e.g., memory, response time, accuracy, I/O cost).

Motivated by the above observations, we explore planning and learning techniques to *automatically induce adaptive data-driven search strategies* within the TA framework to optimize the performance of top-k graph query processing.

Algorithm 2 L2P for TA Framework

Input: Q = graph query, G = data graph, Π_{select} = selection policy, Π_{fetch} = fetching policy

$s \leftarrow I(Q)$ // *initial state*
TERMINATE $\leftarrow False$
while not TERMINATE **do**
  $a_{select} \leftarrow \Pi_{select}(s)$ // *select the sub-query*
  if $a_{select} == HALT$ **then**
    TERMINATE $= True$
  else
    $a_{fetch} \leftarrow \Pi_{fetch}(s, a_{select})$ // *how many to fetch*
    $s \leftarrow$ Apply (a_{select}, a_{fetch}) on s
  end if
end while
return Top-k answers $\mathcal{A}(s)$ corresponding to s

Problem Setup

We assume the availability of a training set of query-answer pairs $\{(Q, A^*)\}$ drawn from an unknown target distribution $\mathcal{D}$, where Q is a graph query and A^* is the corresponding output answer produced by a TA style algorithm $\mathcal{P}$ on data graph G. The accuracy of a query plan (sequence of computational steps for query processing) can be measured using a non-negative loss function L such that $L(Q, \hat{A}, A^*)$ is the loss associated with processing a particular input query Q to produce answer $\hat{A}$ when the correct answer is A^*. The goal of learning is to quickly produce high-quality query plans for minimizing the response time to find correct answers for input queries drawn from the target distribution $\mathcal{D}$.

In this work, we formulate the graph query planning problem in a search-based learning framework. There are three key elements in this framework: 1) the *Search space* $\mathcal{S}_p$ whose states correspond to candidate computational states within the TA framework; 2) the *Selection policy* Π_{select} that is used to select the sub-query to fetch partial matches at each state; and 3) the *Fetching policy* Π_{fetch} that is used to decide how many partial matches to fetch for the selected sub-query at each state.

Generic Search Space. $\mathcal{S}_p$ is a 2-tuple $\langle I, A \rangle$, where I is the initial state function, A gives the set of possible actions in a given state. In our case, $s_0 = I(Q)$ corresponds to a state with one empty list for each sub-query decomposition of Q (say T decompositions without loss of generality). $A(s_i)$ consists of actions that correspond to candidate computational steps (or equivalently search operators for query plans) at state s_i. Each action a is of the

Algorithm 3 Target Query Plan Computation via Beam Search

Input: (Q, A^*) = graph query and answer pair, G = data graph, (I, A) = Search space, b = beam width, H = weighted heuristic function

$B \leftarrow s_0 = I(Q)$ *// Initial state*
TERMINATE $\leftarrow False$
while not TERMINATE **do**
  $C \leftarrow \emptyset$ *// Candidate set*
  for each state $s \in B$ **do**
    if $\mathcal{A}(s) == A^*$ **then**
      TERMINATE $= True$
      $s^* = s$
    else
      Expand s and add all next states to C *// Expansion*
    end if
  end for
  $B \leftarrow$ Top-b scoring states in C via heuristic function $H(s)$ *// Pruning*
end while
return state-action pair sequence from s_0 to s^*

form (i, δ) and corresponds to fetching δ additional partial matches for sub-query i, where $i \in \{1, 2, \cdots, T\} \cup HALT$ and $\delta \in [\delta_{min}, \delta_{max}]$. When *HALT* action is chosen at state s_i (terminal state), we stop the search and return the top-k answer $\mathcal{A}(s_i)$.

We focus on greedy search. The decision process for producing query plans corresponds to choosing a sequence of actions leading from the initial state using both Π_{select} and Π_{fetch}, until Π_{select} selects the *HALT* action (terminal state). Π_{select} and Π_{fetch} are parametrized by feature functions ψ_1 and ψ_2. We want to learn the parameters of both Π_{select} and Π_{fetch} using the training queries, with the goal of quickly producing query plans for minimizing the response time to produce correct answers on unseen queries drawn from $\mathcal{D}$.

Learning to Plan (L2P) Framework

Inspired by the recent success of imitation learning approaches for solving sequential decision-making tasks (Weiss and Taskar 2013; He, Daumé III, and Eisner 2014; 2013; Pinto and Fern 2014), we formulate and solve the problem of learning policies to improve the speed of producing high-quality query plans in the framework of imitation learning.

Overview of Imitation Learning. In traditional imitation learning, expert demonstrations are provided as training data (e.g., demonstrations of a human expert driving a car), and the goal of the learner is to learn to imitate the behavior of an expert performing a task in a way that generalizes to similar tasks or situations. Typically this is done by collecting a set of trajectories of the expert's behavior on a set of training tasks. Then supervised learning is used to find a policy that can replicate the decisions made on those trajectories. Often the supervised learning problem corresponds to learning a classifier or regressor to map states to actions, and off-the-shelf tools can be used.

The two main challenges in applying imitation learning framework to query planning are: 1) Obtaining a high-quality oracle policy that will provide the supervision for the imitation learning process (*oracle construction*); and 2) Learning search control policies that can make search decisions in a computationally-efficient manner (*fast and accurate decision-making*). We provide solutions for these two challenges below.

Oracle Construction

In this section, we describe a generic procedure to compute high-quality query plans for TA framework. We will start by defining the needed terminology.

1) Terminal state. For a given query-answer pair (Q, A^*), a state s in the search space $\mathcal{S}_p$ is called a terminal state if the corresponding top-k answer $\mathcal{A}(s)$ is same as the one (i.e., A^*) produced by running the given TA style algorithm $\mathcal{P}$ on data graph G.

2) Target query plan. For a given query Q, a sequence of actions from initial state to terminal state, $(s_0, a_0), \cdots, (s_N, \varnothing)$, is called a target query plan (TQP), where s_0 is the initial state and s_N is the terminal state.

3) Quality of a target query plan. For a given query Q, the quality of a TQP is defined as the computation time taken to execute the query plan to produce the top-k answer. In other words, quality corresponds to speedup *w.r.t.* computation time of TA style algorithm $\mathcal{P}$ on the same query.

The goal of oracle construction is to compute high-quality target query plans for each training query-answer pair (Q, A^*). A naive approach would be to perform depth-bounded exhaustive search in search space $\mathcal{S}_p$ instantiated for each training query. Since the naive approach is impractical, we resort to heuristic guided beam search and define effective heuristics to make the search efficient.

Heuristics. We define multiple heuristics that can potentially guide the search to uncover high-quality target query plans. These heuristics are defined as a function of the given search state s and the terminal state s^* of the TA style algorithm $\mathcal{P}$. We define the following three concrete heuristics noting that our approach allows to add additional heuristics as needed: **H1)** Total computation time to reach state s from initial state $s_0 = I(Q)$ normalized *w.r.t.* time taken by the TA style algorithm $\mathcal{P}$; **H2)** The cost (or score) of answer $\mathcal{A}(s)$ normalized *w.r.t.* the score of A^* (Arai et al. 2007); and **H3)** The cumulative difference between the size of list (with partial matches) for each sub-query from states s and s^* normalized *w.r.t.* the total size of all lists for s^*.

Target Query Plan Computation via Beam Search. We propose to combine multiple heuristics by their weighted linear combination: $H(s)$=$\sum_{i=1}^{m} w_i \cdot H_i(s)$. For a given weight vector $w \in \Re^m$, the heuristic function is fully specified. For a given training query-answer pair (Q, A^*), we perform breadth-first beam search with beam width b starting at initial state $s_0 = I(Q)$, until we uncover a terminal state (i.e., $\mathcal{A}(s)$ is same as A^*). The sequence of actions leading from s_0 to terminal state s is identified as the target query plan for Q (see Algorithm 3).

Computing Weights via Bayesian Optimization. We don't know the appropriate weights $w_1, w_2, \cdots, w_m$ that will improve the effectiveness of the weighted heuristic function H. We define the value of a candidate weight vector $w \in \Re^m$

over a set of training queries $\mathcal{T}$, $\mathcal{V}(w, \mathcal{T})$, as the average quality of the identified target query plans (via Algorithm 3) for all queries in $\mathcal{T}$ with the corresponding heuristic H. Our goal is to find the weight vector $w \in \Re^m$ that maximizes $\mathcal{V}(w, \mathcal{T})$:

$$w^* = \arg max_{w \in \Re^m} \mathcal{V}(w, \mathcal{T}) \quad (2)$$

The main challenge in solving this optimization problem is that evaluating the value of a candidate weight vector is computationally-expensive. In recent years, Bayesian Optimization (BO) (Shahriari et al. 2016) has emerged as a very popular framework for solving optimization problems where the function evaluations are computationally expensive (e.g., hyper-parameter tuning of machine learning algorithms). Therefore, we propose to find the best weights via BO tools. In short, BO algorithms build a statistical model based on the past function evaluations; and execute a sequential decision-making process that intelligently explores the solution space guided by the statistical model, to quickly reach a near-optimal solution.

Learning Greedy Policies via Imitation Learning

Our goal is to learn a greedy policy $(\Pi_{select}, \Pi_{fetch})$ that maps states to actions in order to imitate the target query plans computed via oracle construction. We assume that for any training query-answer pair (Q, A^*), we can get the oracle query plan $(s_0^*, a_0^*), (s_1^*, a_1^*), \cdots, (s_N^*, \varnothing)$ via Algorithm 3, where s_0^* is the initial state and s_N^* is the terminal state. The goal is to learn the parameters of Π_{select} and Π_{fetch} such that at each state s_i^*, $a_i^* \in A(s_i^*)$ is selected.

Algorithm 4 Learning Greedy Policy via Exact Imitation

Input: $\mathcal{T}$ = Training data
1: Initialize the set of classification examples $\mathcal{D}_1 = \emptyset$
2: Initialize the set of regression examples $\mathcal{D}_2 = \emptyset$
3: **for** each training example $(Q, A^*) \in \mathcal{T}$ **do**
4: Compute the target query plan $(s_0^*, a_0^*), \cdots, (s_N^*, \varnothing)$
5: **for** each search step $t = 0$ to N **do**
6: Generate classification example C_t and regression example R_t to imitate (s_t^*, a_t^*)
7: Aggregate training data: $\mathcal{D}_1 = \mathcal{D}_1 \cup C_t$ and $\mathcal{D}_2 = \mathcal{D}_2 \cup R_t$
8: **end for**
9: **end for**
10: Π_{select} = Classifier-Learner($\mathcal{D}_1$)
11: Π_{fetch} = Regression-Learner($\mathcal{D}_2$)
12: **return** greedy policy $(\Pi_{select}, \Pi_{fetch})$

Exact Imitation Approach. At each state s_t^* on the target path of a training example (Q, A^*), we create one classification example with $\psi_1(s_t^*)$ as input and $a_t^*(1)$ (selection action) as output; and one regression example with $\psi_2(s_t^*, a_t^*(1))$ as input and $a_t^*(2)$ (fetching action) as output (Khardon 1999). The sets of aggregate classification and regression imitation examples collected over all the training examples are then fed to a classifier and regression learner pair, to learn the parameters of Π_{select} and Π_{fetch} (see Algorithm 4). This reduction allows us to leverage powerful off-the-shelf classification and regression learners.

In theory and practice, policies learned via exact imitation can be prone to error propagation: errors in the previous state may result in a next state that is very different from the distribution of states the learner has seen during the training, and contributes to more errors. To mitigate error-propagation problem, we can employ an advanced learning approach like DAgger (Ross, Gordon, and Bagnell 2011).

DAgger Algorithm. The key idea behind DAgger is to generate additional training data so that the learner is able to learn from its mistakes. DAgger is an iterative algorithm, where each iteration adds imitation data to an aggregated data set. The first iteration follows the exact imitation approach. After each iteration, we learn policy $(\Pi_{select}, \Pi_{fetch})$ using the current data. Subsequent iterations perform query planning using the learned policy to generate a trajectory of states for each training query. At each decision along this trajectory, we add a new imitation example if the search decision of the learned policy is different from the oracle policy. In the end, we select the best policy over all the iterations via performance on validation data. To select among several imperfect policies with varying speed and accuracy performance, we pick the policy with the highest accuracy. This principle is aligned with our learning objective.

Handling General Query Planning

In this section, we provide some discussion on ways to extend our L2P framework to more general query planning going beyond the TA style query processing.

There are three key elements in the L2P framework: 1) Search space over query plans; 2) Policy for producing query plans; and 3) Oracle query plans to drive the learning process. The search operators for query plans are specific to each query processing approach. We need to consider additional search operators (e.g., different δ values in the TA framework) to be able to construct high-quality candidate query plans in the search space. The form of the policy will depend on the search operators or actions at search states (e.g., classifier-regressor pair to select the sub-query and number of partial matches to fetch in the TA framework). For computing the oracle plans needed to learn the policy, heuristic-guided beam search is a generic approach, but we need to define effective heuristics as applicable for the given query evaluation approach. A more generic off-the-shelf alternative is to consider *Approximate Policy Iteration* (API) algorithm (Fern, Yoon, and Givan 2006). API starts with a default policy and iterates over the following two steps. **Step 1:** Generate trajectories of current rollout policy from initial state; and **Step 2:** Learn a fast approximation of rollout policy via supervised learner (e.g., classifier) to induce a new policy. Indeed, each iteration of API can be seen as imitation learning, where trajectories of current rollout policy correspond to expert demonstrations and the new policy is induced using the exact imitation algorithm (Fern 2016).

L2P Instantiation for STAR

In this section, we provide a concrete instantiation of our L2P framework for STAR, a state-of-the-art graph query reasoner based on the TA framework.

Overview of STAR Query Processing. STAR is an instantiation of TA framework, where the graph query is decomposed into a set of star-shaped queries. A very recent work (Yang et al. 2016) showed both theoretically and empirically that star decomposition provides a good trade-off between sub-query evaluation cost and the number of candidate partial answers. In particular, partial answers for each star query can be efficiently generated on demand with a guaranteed sorted access as per the given scoring function $F(\cdot)$.

Example 2: Figure 2 illustrates STAR query processing to find the top-1 answer for the query Q shown in Figure 1. It first decomposes Q into two stars (Q_1^* and Q_2^*), i.e., graphs with a unique pivot and one or more adjacent nodes. It then fetches partial answers for each star query in a sorted manner, and joins the partial answers whenever possible, until it finds a complete match and termination criteria is met. □

Search Space. Suppose the given graph query Q is decomposed into a set of star-queries $\{Q_1^*, \ldots, Q_T^*\}$ without loss of generality. The initial state $s_0 = I(Q)$ corresponds to a state with one empty list for each star query decomposition of Q. Recall that each action a (i.e., search operator for query plan) is of the form (i, δ) and corresponds to fetching δ additional partial matches for star query i, where $\delta \in [\delta_{min}, \delta_{max}]$. We employed $\delta_{min} = 10$ and $\delta_{max} = 200$ (candidate number of partial matches to fetch), and considered candidate choices in the multiples of δ_{min}, i.e., $\frac{\delta_{max}}{\delta_{min}}$ discrete values. This choice is mainly driven by the computational complexity of finding oracle query plans via breadth-first beam search. The branching factor at each search step is $b \times T \times \frac{\delta_{max}}{\delta_{min}}$, where b is the beam width. Smaller values of δ_{min} may lead to higher quality query plans, but only at the expense of increased computational complexity.

Features. To drive the learning process, we define feature functions ψ_1 and ψ_2 over search states. Our features can be categorized into three groups: 1) *Static features* that are computed from the query topology, decomposed star queries, and initial partial matches. Some examples include the number of nodes and edges in each star query, the number of candidate nodes in data graph, and the number of joinable nodes in a star decomposition; 2) *Ranking features* that are computed from the lower bound of current top-k answers and upper bound for each star query; and 3) *Context features* that are computed based on the current state of L2P-STAR like the selected star query and the total number of fetches for each star query. The features are cheap to compute. See Appendix for complete details of features.

Experiments and Results

Using three large real-world knowledge graphs, we empirically evaluate the performance of our instantiation of L2P framework for STAR query processing approach.

Experimental Setup

Datasets. We employ three real-world open knowledge graphs: 1) YAGO (`mpi-inf.mpg.de/yago`) contains $2.6M$ entities (*e.g.,* people, companies, cities), $5.6M$ relationships, and $10.7M$ nodes and edge labels, extracted from several public knowledge bases including Wikipedia; 2) DBpedia (`dbpedia.org`) consists of $3.9M$ entities, $16.8M$ edges, and $14.9M$ labels; and 3) Freebase (`freebase.com`), a more diversified knowledge graph that contains $40.3M$ entities, $180M$ edges, and $81.6M$ labels.

Yago	DBPedia	Freebase
590s	680s	1120s

Table 1: Average runtime (in seconds) of Algorithm 3 with beam width b=10.

Query Workload. We developed a query generator to produce both training and testing queries following the *DBPSB* benchmark (Morsey et al. 2011). We first generate a set of query templates, each has a topology sampled from a graph category (de Ridder et al), and is assigned with labels (specified as "entity types") sampled from top 20% most frequent labels in the real-world graphs. We created 20 templates to cover common entity types, and generated a total of $2K$ queries by instantiating the templates. We employ 50% queries for training, 20% for validation, and 30% for testing respectively.

STAR Implementation. We implemented the STAR query processing framework (Yang et al. 2016) in Java using the Neo4j (`neo4j.com`) graph database system.

Oracle Policy Implementation. We performed heuristic guided breadth-first beam search with different beam widths $b = \{1, 5, 10, 20, 50\}$ to compute high-quality target query plans for each training query (see Algorithm 3). BayesOpt software (Martinez-Cantin 2014) was employed to find the weights of heuristics with expected improvement as the acquisition function. We did not see noticeable performance improvement beyond 100 iterations. Since we didn't get significant speedup improvements beyond $b = 10$, we employed target query plans (aka Oracle policy) obtained with $b = 10$ for all our training and testing experiments.

L2P-STAR Implementation. We employed XGBoost (Chen and Guestrin 2016), an efficient and scalable implementation of functional gradient tree boosting for classification and regression, as our base learner. All hyperparameters (boosting iterations, tree depth, and learning rate) were automatically tuned based on the validation data using BayesOpt (Martinez-Cantin 2014), a state-of-the-art BO software. L2P-STAR (Exact) and L2P-STAR (DAgger) corresponds to policy learning via exact imitation and DAgger algorithms respectively. We performed 5 iterations of DAgger and selected the policy with the highest accuracy on the validation data. L2P-STAR needs to additionally store the learned policy (classifier and regressor pair). However, this overhead is negligible when compared to the memory usage of Neo4j.

Code and Data. All the code and data related to this work is publicly available on a GitHub repository. We will provide the link when this research is published.

Evaluation Metrics. We evaluate STAR, Oracle, and L2P-STAR using the following two metrics.

1) *Speedup.* For a given query Q, the response time of an algorithm $\mathcal{P}$, denoted as $\texttt{Time}(\mathcal{P}, Q)$, refers to the total CPU

	YAGO			DBpedia			Freebase		
	speedup	accuracy	run-time (ms)	speedup	accuracy	run-time (ms)	speedup	accuracy	run-time (ms)
STAR	1.00	100%	1925.78	1.00	100%	6401.48	1.00	100%	13932.26
Oracle	4.53	100%	757.70	5.62	100%	1192.26	62.53	100%	1478.60
L2P-STAR (Exact)	3.66	93%	764.00	5.00	97%	1310.26	47.69	95%	1550.03
L2P-STAR (DAgger)	3.71	94%	804.61	5.00	97%	1310.26	47.69	95%	1550.03

Table 2: Speedup, accuracy, and query run-time results (averaged over testing queries) comparing STAR, Oracle, and L2P-STAR.

	DBpedia					Freebase				
	Analytical Ops		Computation Time (ms)			Analytical Ops		Computation Time (ms)		
	# Fetch	# Join	Fetch	Join	ML-Overhead	# Fetch	# Join	Fetch	Join	ML-Overhead
STAR	3,876	13,106	3,101.51	3,283.58	N/A	5,732	20,048	3,868.65	10,044.39	N/A
Oracle	28	43	1,208.89	6.32	N/A	35	65	1,487.12	7.64	N/A
L2P-STAR	80	134	1,238.89	49.69	23.01	72	122	1,512.76	19.64	17.70

Table 3: Statistics of different analytical operations (e.g., fetch and join calls) and the corresponding computation time (averaged over testing queries).

time taken from receiving the query to finding the top-k answer. The speedup factor $\tau(\mathcal{P}, Q)$ is computed as the ratio of $\texttt{Time}(\mathsf{STAR}, Q)$ to $\texttt{Time}(\mathcal{P}, Q)$.

2) *Accuracy.* We employ Levenshtein distance function (Navarro 2001) to compute node/edge label similarity, and R-WAG's ranking function(Roy, Eliassi-Rad, and Papadimitriou 2015) to compute the matching score $F(\cdot)$. For a given query Q, the accuracy of an algorithm $\mathcal{P}$ is defined as the ratio of the matching score of correct answer A^* to the matching score of predicted answer $\hat{A}$.

We conducted all our experiments on a Windows server with 3.5 GHz Ci7 CPU and 32GB RAM configuration. All experiments are conducted 3 times and averaged results are presented. We report the average metrics (speedup, accuracy, and computation time) over all testing queries.

Results. We organize our results along different dimensions.

Oracle Policy Computation Time. Table 1 shows the average time to compute the oracle query plan via beam search with beam width b=10 (see Algorithm 3). The runtime is high and it increases for denser graphs (e.g., Freebase). Hence, we cannot use this computationally expensive procedure in real-time and need L2P to quickly generate high-quality query plans.

STAR **vs. Oracle.** To find out the overall room for improvement via L2P framework, we compare STAR with the oracle policy. Recall that the oracle policy for a given query Q, corresponds to the target query plan obtained by heuristic-guided breadth-first beam search, and relies on the knowledge of correct answer A^*. Table 2 shows the speed, accuracy, and run-time; and Table 3 shows the statistics of different analytical operations (e.g., fetch and join calls) and the corresponding computation time.

We make the following observations: 1) There is significant room for improving STAR via L2P framework as noted by speedup of oracle policy for different datasets (4.53 for YAGO, 5.62 for DBpedia, and 62.53 for Freebase); 2) The speedup of oracle policy is higher for large and dense data graphs, *e.g.,*Freebase. Indeed, for dense graphs, we expect more candidate matches for each star-query, which can make the STAR very slow and L2P-STAR would be more beneficial. In fact, there is a growing evidence that the real-world graphs become denser over time and follows a power-law pattern (Leskovec, Kleinberg, and Faloutsos 2007); and 3) The number of fetch/join calls can be reduced by two orders of magnitude by following the plan from oracle policy. As pointed out earlier, one of the major drawbacks of TA style STAR is that it fetches many "useless" partial answers that do not contribute to top-k answers. The performance of oracle policy shows that it is possible to significantly improve STAR if the learner can successfully imitate the oracle plans.

L2P-STAR **vs. Oracle.** We compare L2P-STAR with the oracle policy to understand how well the learner is able to mimic the search behavior of oracle. We make the following observations from Table 2. The speed and accuracy of L2P-STAR in general is very close to the oracle policy across all the datasets. L2P-STAR loses at most 6% accuracy and significantly improves the speed when compared to STAR. For example, L2P-STAR improves the speed of STAR by 47 times with an accuracy loss of 5% for Freebase. L2P-STAR has to learn when to stop the search (i.e., select *HALT* action). If L2P-STAR stops the search early, it will lose accuracy. Similarly, it will lose speed when stopping of the search is delayed. From Table 3, we can see that the overhead of L2P-STAR to make search decisions (i.e., computing features and excuting the classifier/regressor) is very small when compared to the query processing time (2% of query run-time on an average).

We did not see performance improvement in L2P-STAR by training with DAgger when compared to training with exact-imitation (except for YAGO). This is due to our principle of selecting among multiple imperfect policies: pick the policy with highest accuracy. We were able to uncover policies with higher speedup over exact imitation, but their accuracy was relatively low.

Ablation Analysis. STAR, and L2P-STAR have their corresponding selection and fetching policies. To understand how the learned selection and fetching policies Π_{select} and Π_{fetch} affect STAR individually, we plug them one at a time. Table 4 shows the results of this analysis for all the three datasets. If we employ Π_{fetch} for adaptive fetching inside STAR, we get a speedup of 1.86, 2.19, and 4.78 for YAGO, DBpedia, and Freebase respectively, without losing any accuracy. Therefore, when practitioner requires 100%

		Expansion					
		YAGO		DBpedia		Freebase	
		STAR	L2P-STAR	STAR	L2P-STAR	STAR	L2P-STAR
Selection	STAR	(1, 100%)	(1.86, 100%)	(1, 100%)	(2.19, 100%)	(1, 100%)	(4.78, 100%)
	L2P-STAR	(4.08, 87%)	(3.71, 94%)	(5.81, 90%)	(5.00, 97%)	(54.14, 83%)	(47.69, 95%)

Table 4: Results of ablation analysis.

accuracy, this combination can be deployed. This also shows the usefulness of Π_{fetch} alone. However, the overall performance of STAR with Π_{fetch} alone is much worse than L2P-STAR and shows the importance of Π_{select}.

		Test dataset		
		YAGO	DBpedia	Freebase
Train	YAGO	**(3.71, 0.93)**	(3.88, 0.96)	(22, 0.92)
	DBpedia	(4.30, 0.90)	**(5, 0.97)**	(55.36, 0.89)
	Freebase	(4.24, 0.88)	(5.72, 0.89)	**(47.69, 0.95)**
	Combined	(3.87, 0.92)	(4.27, 0.97)	(55.84, 0.96)

Table 5: Transfer learning results. Table cells contain the speedup and accuracy pair of L2P-STAR for different train and test configurations.

Transfer Learning. The learned policy can be seen as a function that maps search states to appropriate actions via features. Since the feature definitions are general, one could hypothesize that the learned knowledge is general and can be used to query different data graphs. To test this hypothesis, we learned policies on each of the three datasets, another policy using the combination of all the three datasets; and tested each policy on both individual datasets and the combined dataset. We make the following observations from Table 5. The learned policies generalize reasonably well to datasets that are not used for their training. We get the best accuracy when training and testing are done on the same dataset. The only exception is that the policy trained on the combined dataset gave better performance on Freebase.

Change %	(Speedup, Accuracy)
0%	(4.12, 95.9%)
5%	(5.26, 93.2%)
10%	(7.03, 93.1%)

Table 6: Performance of L2P-STAR with changes to the trained data graph.

Changing Data Graph. To investigate the stability of the learned policy with changes to the training data graph, we performed some experiments on DBpedia graph. We do not have access to the temporal data graph. Therefore, we created multiple samples of original data graph with varying sizes by employing the well-studied *Forest Fire (FF)* approach (Leskovec and Faloutsos 2006). We train on the smallest data sample (90% of the original graph) and test on larger samples. This setup is based on the fact that real-world graphs are known to become large and dense over time (Leskovec, Kleinberg, and Faloutsos 2007). From Table 6, we can see that by changing 10% of the training data graph (i.e., 1.6M additional edges), L2P-STAR loses at most 3% accuracy (similar to transfer learning results). Additionally, the speedup of L2P-STAR improves as the data graph evolves. As explained before, it is natural to expect more speedup with large and dense graphs: increased candidate matches for each sub-query can make STAR slower and L2P-STAR can be more beneficial. Indeed, Table 2 corroborates this hypothesis *e.g.,* Freebase.

In general, we need to update the policy whenever the distribution of queries and/or data graph changes significantly. A thorough investigation of this aspect is part of our immediate future work.

Related Work

Our work is related to a sub-area of AI called speedup learning (Fern 2010). Specifically, it is an instance of inter-problem speedup learning. Reinforcement learning (RL), imitation learning (IL), and hybrid approaches combining RL and IL have been explored to learn search control knowledge from training problems in the context of diverse application domains. Some examples include job shop scheduling (Zhang and Dietterich 1995), deterministic and stochastic planning (Xu, Fern, and Yoon 2009; Pinto and Fern 2014), natural language processing (Jiang et al. 2012; He, Daumé III, and Eisner 2012; 2013), computer vision (Weiss, Sapp, and Taskar 2013; Weiss and Taskar 2013), and mixed-integer programming solvers (He, Daumé III, and Eisner 2014; Khalil et al. 2016). Our work explores this speedup learning problem for query planning in graph databases, a novel application domain. We formalized and solved this problem in a learning to plan framework. There is some work on applying learning in the context of query optimization (Hasan and Gandon 2014; Ganapathi et al. 2009; Gupta, Mehta, and Dayal 2008), but we are not aware of any existing work that tightly integrates learning and search for graph query planning as done in this work.

Knoblock and Kambhampati has done seminal work on applying automating planning techniques for information integration on the web (Knoblock and Kambhampati 2007). Their work leverages the relationship between query planning in information integration and automated planning with sensing (i.e., information gathering) actions. Our work is different from theirs as we explore query planning in the context of traditional (graph) databases and rely heavily on advanced learning techniques.

Summary and Future Work

We developed a general learning to plan (L2P) framework to improve the computational efficiency of a large-class of query reasoners that follow the Threshold Algorithm framework. We integrated learning and search to generate adaptive query plans in a data-driven manner via training queries. We showed that our concrete instantiation L2P-STAR can achieve significant speedup over STAR with negligible loss in accuracy across multiple knowledge graphs. Future work includes exploring ways to further improve the accuracy of L2P-STAR without losing speedup; scaling up L2P framework to handle large number of batch/streaming queries by exploring active learning techniques; and deploying L2P instantiations for both relational and graph databases in real-world applications.

References

Arai, B.; Das, G.; Gunopulos, D.; and Koudas, N. 2007. Anytime measures for top-k algorithms. In *VLDB*.

Chen, T., and Guestrin, C. 2016. Xgboost: A scalable tree boosting system. In *KDD*.

Cheng, J.; Zeng, X.; and Yu, J. X. 2013. Top-k graph pattern matching over large graphs. In *ICDE*.

Chin Jr, G.; Choudhury, S.; Feo, J.; and Holder, L. 2014. Predicting and detecting emerging cyberattack patterns using streamworks. In *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, 93–96.

Corporation, N. 2013. Information system on graph classes and their inclusions (isgci). *http://docs.neo4j.org/chunked/stable/cypher-query-lang.html*.

Das, M.; Wu, Y.; Khot, T.; Kersting, K.; and Natarajan, S. 2016. Scaling lifted probabilistic inference and learning via graph databases. In *SDM*.

de Ridder et al, H. Information system on graph classes and their inclusions (isgci). *graphclasses.org*.

Ding, X.; Jia, J.; Li, J.; Liu, J.; and Jin, H. 2014. Top-k similarity matching in large graphs with attributes. In *DASFAA*.

Fagin, R.; Lotem, A.; and Naor, M. 2003. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences* 66(4):614–656.

Fern, A.; Yoon, S. W.; and Givan, R. 2006. Approximate policy iteration with a policy language bias: Solving relational markov decision processes. *JAIR* 25:75–118.

Fern, A. 2010. Speedup learning. In *Encyclopedia of Machine Learning*.

Fern, A. 2016. Personal Communication.

Ganapathi, A.; Kuno, H.; Dayal, U.; Wiener, J. L.; Fox, A.; Jordan, M.; and Patterson, D. 2009. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *ICDE*.

Gupta, C.; Mehta, A.; and Dayal, U. 2008. Pqr: Predicting query execution times for autonomous workload management. In *ICAC*.

Hasan, R., and Gandon, F. 2014. A machine learning approach to sparql query performance prediction. In *IEEE/WIC/ACM*.

He, H.; Daumé III, H.; and Eisner, J. 2012. Imitation learning by coaching. In *NIPS*.

He, H.; Daumé III, H.; and Eisner, J. 2013. Dynamic feature selection for dependency parsing. In *EMNLP*.

He, H.; Daumé III, H.; and Eisner, J. 2014. Learning to search in branch and bound algorithms. In *NIPS*.

Jiang, J.; Teichert, A. R.; Daumé III, H.; and Eisner, J. 2012. Learned prioritization for trading off accuracy and speed. In *NIPS*.

Khalil, E. B.; Bodic, P. L.; Song, L.; Nemhauser, G. L.; and Dilkina, B. N. 2016. Learning to branch in mixed integer programming. In *AAAI*.

Khardon, R. 1999. Learning to take actions. *Machine Learning* 35(1):57–90.

Kim, J.; Shin, H.; Han, W.-S.; Hong, S.; and Chafi, H. 2015. Taming subgraph isomorphism for rdf query processing. *VLDB* 1238–1249.

Knoblock, C., and Kambhampati, S. 2007. Tutorial on information integration on the web. In *AAAI*.

Leskovec, J., and Faloutsos, C. 2006. Sampling from large graphs. In *SIGKDD*.

Leskovec, J.; Kleinberg, J.; and Faloutsos, C. 2007. Graph evolution: Densification and shrinking diameters. *TKDD* 2.

Lu, J.; Lin, C.; Wang, W.; Li, C.; and Wang, H. 2013. String similarity measures and joins with synonyms. In *SIGMOD*.

Martinez-Cantin, R. 2014. Bayesopt: A bayesian optimization library for nonlinear optimization, experimental design and bandits. *Journal of Machine Learning Research* 15:3915–3919.

Morsey, M.; Lehmann, J.; Auer, S.; and Ngonga Ngomo, A.-C. 2011. DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data. In *ISWC*.

Navarro, G. 2001. A guided tour to approximate string matching. *CSUR* 33(1):31–88.

Niu, F.; Ré, C.; Doan, A.; and Shavlik, J. W. 2011. Tuffy: Scaling up statistical inference in markov logic networks using an RDBMS. *PVLDB* 4(6):373–384.

O'Brien, T.; Ritz, A.; Raphael, B.; and Laidlaw, D. 2010. Gremlin: an interactive visualization model for analyzing genomic rearrangements. *TVCG* 918–926.

Pinto, J., and Fern, A. 2014. Learning partial policies to speedup MDP tree search. In *UAI*.

Prud'Hommeaux, E.; Seaborne, A.; et al. 2008. Sparql query language for rdf. *W3C recommendation* 15.

Ross, S.; Gordon, G. J.; and Bagnell, D. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *AISTATS*.

Roy, S. B.; Eliassi-Rad, T.; and Papadimitriou, S. 2015. Fast best-effort search on graphs with multiple attributes. *TKDE* 27(3):755–768.

Sarkhel, S.; Venugopal, D.; Pham, T. A.; Singla, P.; and Gogate, V. 2016. Scalable training of markov logic networks using approximate counting. In *AAAI*.

Shahriari, B.; Swersky, K.; Wang, Z.; Adams, R. P.; and de Freitas, N. 2016. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE* 104(1):148–175.

Weiss, D. J., and Taskar, B. 2013. Learning adaptive value of information for structured prediction. In *NIPS*.

Weiss, D. J.; Sapp, B.; and Taskar, B. 2013. Dynamic structured model selection. In *ICCV*.

Xu, Y.; Fern, A.; and Yoon, S. W. 2009. Learning linear ranking functions for beam search with application to planning. *JMLR* 10:1571–1610.

Yang, S.; Han, F.; Wu, Y.; and Yan, X. 2016. Fast top-k search in knowledge graphs.

Zeng, X.; Cheng, J.; Yu, J.; and Feng, S. 2012. Top-k graph pattern matching: A twig query approach. *WAIM*.

Zhang, W., and Dietterich, T. G. 1995. A reinforcement learning approach to job-shop scheduling. In *IJCAI*.

Zhang, C.; Kumar, A.; and Ré, C. 2016. Materialization optimizations for feature selection workloads. *ACM TODS* 41(1):2.

Zou, L.; Huang, R.; Wang, H.; Yu, J. X.; He, W.; and Zhao, D. 2014. Natural language question answering over RDF: a graph data driven approach. In *SIGMOD*.