Integrating Connection Search in Graph Queries

Angelos Christos Anadiotis, Ioana Manolescu, Madhulika Mohanty

Integrating connection search in graph queries

Angelos Christos Anadiotis*
Oracle, Switzerland
angelos.anadiotis@oracle.com

Ioana Manolescu
Inria and IPP, France
ioana.manolescu@inria.fr

Madhulika Mohanty
Inria and IPP, France
madhulika.mohanty@inria.fr

ABSTRACT

Graph data management and querying has many practical applications. When graphs are very heterogeneous and/or users are unfamiliar with their structure, they may need to *find how two or more groups of nodes are connected in a graph*, even when users are not able to describe the connections. This is only partially supported by existing query languages, which allow searching for *paths*, but not for *trees connecting three or more node groups*. The latter is related to the NP-hard Group Steiner Tree problem, and has been previously considered for keyword search in databases.

In this work, we formally show how to integrate *connecting tree patterns* (CTPs, in short) within a graph query language such as SPARQL or Cypher, leading to an *Extended Query Language* (or EQL, in short). We then study a set of algorithms for evaluating CTPs; we generalize prior keyword search work, most importantly by (*i*) considering bidirectional edge traversal and (*ii*) allowing users to select *any* score function for ranking CTP results. To cope with very large search spaces, we propose an efficient pruning technique and formally establish a large set of cases where our algorithm, MoLESP, is complete even with pruning. Our experiments validate the performance of our CTP and EQL evaluation algorithms on a large set of synthetic and real-world workloads.

1 INTRODUCTION

Graph databases are increasingly adopted in a wide range of applications spanning from social network analysis to scientific data exploration, the financial industry, and many more. To query RDF graphs, one can use the W3C's standard SPARQL [13] query language; for property graphs, Cypher [35] is among the best known. An interesting but challenging query language feature is *reachability*: a SPARQL 1.1 query can check, e.g., if there are some paths along which Mr. Shady deposits funds into a given bank ABC. Such queries are important in investigative journalism applications [5], in the fight against money laundering, etc. SPARQL allows checking for the existence of a path, but does not return the matching paths to users. In contrast, a Cypher query may also return the paths between two given sets of nodes.

Unfortunately, none of these languages support finding trees, connecting three (or more) sets of nodes, while the latter can be very useful. For instance, when investigating ill-acquired wealth, one may want to find "all connections between Mr. Shady, bank company ABC, and the tax office of the DEF republic": an answer to this query is a *tree*, connecting three nodes corresponding to the person, bank, and tax office, respectively.

Searching for connections among m sets of nodes is closely related to the Group Steiner Tree Problem (GSTP), which asks for *the least-cost*, e.g., fewest-edges, tree; the problem is NP-hard. The database literature has studied many variants of this problem under the name of *keyword search in databases*, for e.g., [1, 4, 10, 12, 16, 26, 30, 39, 40, 44]. To cope with the high complexity, existing algorithms (*i*) consider a fixed cost function and leverage its properties to limit the search, (*ii*) propose approximate solutions, within a known distance from the optimum, and/or (*iii*) implement heuristics without guarantees but which have performed well on some problems.

Requirements Our recent collaborations with investigative journalists [5, 6] lead to identifying the following set of needs. First, (**R1**) *graph query languages should allow returning trees that connect m node sets*, for some integer $m \geq 2$; (**R2**) it must be possible to search for connecting trees *orthogonally to (or, in conjunction with any) score functions* used to compare and rank the trees. This is because different graphs and applications are best served by different scores, and when exploring a graph, journalists need to experiment with several before they find interesting patterns. For instance, in the example above, if Mr. Shady is a citizen of DEF and ABC has offices there, the smallest solution connects them through the DEF country node; however, this is not interesting to journalists. Instead, a connection through three ABC accounts, sending money from DEF to Mr. Shady in country GHI, is likely much more interesting. An orthogonal requirement is (**R3**) to *treat graphs as undirected when searching for trees*. For instance, the graph may contain "Mr. Shady $\xrightarrow{\text{hasAccount}}$ acct_1", or, just as likely, "acct_1 $\xrightarrow{\text{belongsTo}}$ Mr. Shady". We cannot afford to miss a connecting tree because we "expected" an edge in a direction and it happens to be in the opposite direction. Further, (**R4**) *all answers need to be found (within a time and/or space budget)* for several reasons: (*i*) continuity with the semantics of standard graph query languages, that also return all results (unless users explicitly LIMIT the result size); (*ii*) to remain independent of, and thus orthogonal to, the cost function (recall (R2)); and, (*iii*) for practical reasons, given the problem complexity, which is further exacerbated by (R3), and renders complete search on large graphs unfeasible. Finally, (**R5**) *the extended queries should be efficiently executed*, even when graphs are *highly heterogeneous*, as in investigative journalism scenarios, where text, structured, and/or semistructured sources are integrated together.

Contributions To address the above requirements, we make the following contributions:

(1) We *formally define an Extended Query Language (EQL, in short)*, which combines together Basic Graph Pattern (or conjunctive) queries at the core of both SPARQL and Cypher, and Connecting Tree Patterns (CTPs, in short). A CTP allows searching for trees that connect m groups of nodes, for $m \geq 2$. BGPs and CTPs can be freely joined. This addresses requirements (R1), (R2), and also (R3), since our CTP semantics returns trees regardless of the edge directions (Section 2).

(2) We provide a *scalable EQL query evaluation strategy*, which leverages existing algorithms for the well-studied problem of evaluating conjunctive queries, contributing to (R5) (Section 3).

*Work done while at Ecole Polytechnique.

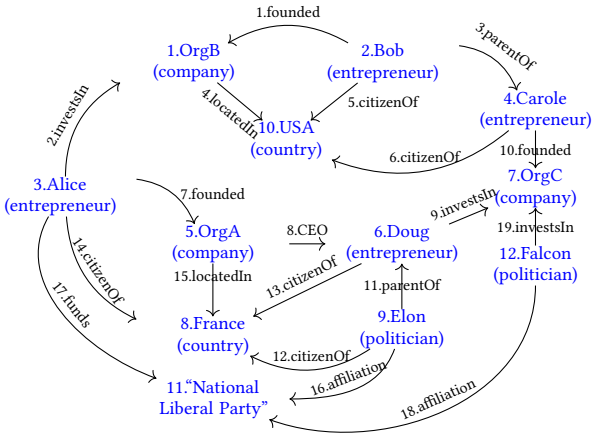


Figure 1: Sample data graph.

(3) For *CTP evaluation*, we study a set of baseline algorithms, and explain that their performance suffers due to repeated (wasted) work and/or the need to minimize the trees they find; GAM [6] algorithm is more efficient, but it does not scale in all cases. We introduce a powerful *Edge Set Pruning (ESP) technique*, which significantly speeds up the execution, but can lead to incompleteness. We then bring two orthogonal modifications which, combined, lead to our MoLESP algorithm, for which we *formally establish completeness for* $m \in \{2, 3\}$, which are most frequent, as well as for *a large class of results for arbitrarily large m*. This addresses requirement (R4) and contributes to (R5) (Section 4).

(4) We experimentally show that: (*i*) baseline algorithms inspired from breadth-first search are unfeasible even for small graphs; (*ii*) the optimizations we bring here over the GAM algorithm [6] strongly reduce the search time; (*iii*) integrating our MoLESP algorithm with a simple conjunctive graph query engine allows to efficiently evaluate queries in our extended language (Section 5).

2 EXTENDED QUERY LANGUAGE (EQL)

Definition 2.1 (Graph)**.** A graph $\mathbf{G}(\mathbf{N}, \mathbf{E})$ consists of a set of nodes $\mathbf{N}$ and a set of edges $\mathbf{E} \subseteq \mathbf{N} \times \mathbf{N}$. Each node $n \in \mathbf{N}$ carries a label $l(n)$ from a label set $\mathbf{L}$, which includes the empty label ϵ. Similarly, each edge $e \in \mathbf{E}$ has a label $l(e) \in \mathbf{L}$.

The two main graph data models are RDF graphs, and property graphs (PGs). To illustrate, in the following, we will rely on RDF graphs; our work can be transposed with only surface changes to PGs. Figure 1 introduces a sample graph, assigning an integer ID and label to each node and edge. We will refer to nodes as n_1, n_2, etc., e.g., n_1 is the node whose ID is 1 and label is OrgB, and similarly to edges as e_1, e_2, etc. Labels of literal nodes, e.g., n_{11}, are enclosed in quotes; the other nodes are URIs.

Node and edge properties Graph nodes and edges may have other properties beyond labels; for instance, an RDF node may have 0 or more *types*. In our example, types are shown in parentheses under the nodes. In a PG, nodes and edges can have multiple properties. We denote by $\mathcal{P}$ the set of all properties that nodes and edges may have; each property $p \in \mathcal{P}$ is a function p that, given a node n (or edge e), returns $p(n)$, the value of property p on node n (and similarly for e). Without loss of generality, we consider that l:N→**L** belongs to $\mathcal{P}$, that is, the label is a node and/or edge property.

Let $\mathcal{V}$ be a set of variable names, to be used in queries. Let $\Omega = \{=, <, \leq, \sim\}$ be a set of comparison operators, where $\sim$ denotes pattern matching such as SQL's like operator. They are used to express predicates over nodes and/or edges, as follows:

Definition 2.2 (Predicate)**.** A *condition* over a variable $v \in \mathcal{V}$ is of the form $p(v)\ op\ c$ where $p \in \mathcal{P}$, $op \in \Omega$ and c is a constant such that the operator op is well-defined on any value of property p together with c. A *predicate over* v is a conjunction of conditions over v. An empty predicate (no conditions) over v is simply v.

A node $n \in \mathbf{N}$ (or edge $e \in \mathbf{E}$) *satisfies the predicate* if and only if, in every condition of the predicate, replacing v with n (respectively, e) and evaluating op yields true. For instance, $l(v)\sim$"*lice" $\wedge$ $\tau(v)=\tau_{\text{entrepreneur}}$ is a predicate consisting of two conditions, one on the label (which must end in the string "lice") and one on the type, which must be entrepreneur. This predicate is true on the node n_3 in our example, and false on the other nodes and edges. Any node or edge satisfies the empty predicate. *For readability, when a predicate consists of exactly an equality between a node or edge label and a constant, we simply use the constant to denote the predicate*, thus, $l(v)$ = "Alice" can be simply written "Alice", when this is unambiguous. However, *each predicate always involves exactly one variable* (v in our example), *even when the short syntax hides it*. We will revert to the longer syntax when we need to make the variable explicit, e.g., use it several times in the query.

Definition 2.3 (Edge Pattern)**.** An edge pattern is a triple (p_1, p_2, p_3) of three predicates: p_1 holds over the source node of an edge, p_2 over the edge itself, and p_3 over the target node.

For instance, $(l(s)$="Alice", $l(e)$="citizenOf", $d)$ states that the source node s is labeled "Alice" and the edge e is labeled "citizenOf". The third predicate is a variable. With the above simplification, we can also write this pattern as ("Alice", "citizenOf", d).

A core construct of graph query languages is:

Definition 2.4 (Basic Graph Pattern)**.** A Basic Graph Pattern (BGP) b is a set of edge patterns that are *connected* in the following sense. If the BGP contains at least 2 edge patterns, each pattern must have a common variable with another edge pattern.

A sample BGP b_1 is: {(x, "citizenOf", "USA"), (x, "founded", "OrgB")}.

Definition 2.5 (CT Pattern)**.** A connecting tree pattern (CTP, in short) is a tuple of the form: $g = (g_1, g_2, \ldots, g_m, \underline{v_{m+1}})$ where each g_i, $1 \leq i \leq m$ is a predicate and $\underline{v_{m+1}}$ is a variable. All variables occurring in $g_1, \ldots, g_m, \underline{v_{m+1}}$ are pairwise distinct.

CTPs are used to find connections among nodes, as follows. When replacing each g_i with a graph node, v_{m+1} is bound to a *subtree* of **G**, having these nodes as leaves (we formalize this below). To visually distinguish BGPs from CTPs, we always underline the last variable of a CTP.

Definition 2.6 (Core query)**.** A core query Q has a *head* and a *body*. The body is a set of k BGPs, $k \geq 0$, and l CTPs, $l \geq 0$, such that $k + l > 0$, and each underlined (last) variable from a CTP appears exactly once in Q. The head is a subset of the body variables.

An example core query, Q_1, consists of 3 BGPs and a CTP:

Q_1: $(x, y, z, \underline{w})$:- $(\tau(x) = \tau_{\text{entrepreneur}}$, "citizenOf", "USA")
$(\tau(y) = \tau_{\text{entrepreneur}}$, "citizenOf", "France"),
$(\tau(z) = \tau_{\text{politician}}$, "citizenOf", "France"), $(x, y, z, \underline{w})$

Q_1 asks: "What are the connections $\underline{w}$ between some American entrepreneur x, some French entrepreneur y, and some French politician z?" We denote the CTP of this query by g^1. To define core query semantics, our first notion is:

Definition 2.7 (BGP embedding)**.** Given a BGP $b = \{t_1, \ldots, t_k\}$, an embedding of b into **G** is a function ϕ, associating to each variable v in b, a node $n \in \mathbf{N}$ or an edge $e \in \mathbf{E}$, such that (*i*) $\phi(v)$ satisfies all the predicates on v in b; and (*ii*) for every edge pattern (s, e, d) in b, the edge $\phi(e) \in \mathbf{E}$ goes from $\phi(s)$ to $\phi(d)$.

A sample embedding ϕ for the first BGP of Q_1 maps x to n_4, "USA" to n_{10}, "citizenOf" to e_6, etc.

Next, we define:

Definition 2.8 (Set-based CTP result)**.** Let $g = (g_1, \ldots, g_m, \underline{v_{m+1}})$ be a CTP pattern and $S_1, \ldots, S_m$ be sets of **G** nodes, called **seed sets**, such that every node in S_i satisfies g_i, for $1 \leq i \leq m$. The *result of g based on $S_1, \ldots, S_m$*, denoted $g(S_1, \ldots, S_m)$, is the set of all $(s_1, \ldots, s_m, t)$ tuples such that $s_1 \in S_1, \ldots, s_m \in S_m$ and t is a *minimal* subtree of **G** containing the nodes $s_1, \ldots, s_m$. By minimal, we mean that (*i*) removing any edge from t disconnects it and/or removes some s_i from t, and (*ii*) t contains only one node from each S_i.

In our sample graph, let $S_1 = \{n_2, n_4\}$ (US entrepreneurs), $S_2 = \{n_3, n_6\}$ (French entrepreneurs), and $S_3 = \{n_9\}$ (French politicians). Then, $g^1(S_1, S_2, S_3)$ includes $(n_4, n_6, n_9, t_\alpha)$ where the tree t_α consists of the edges $n_4 \xrightarrow{e_{10}} n_7 \xleftarrow{e_9} n_6 \xleftarrow{e_{11}} n_9$, also denoted by $\{e_{10}, e_9, e_{11}\}$ for brevity. Another result of this CTP is (n_2, n_3, n_9, t_β), with $t_\beta = \{e_1, e_2, e_{17}, e_{16}\}$. This result is only possible because Def. 2.8 allows trees to span over **G** edges *regardless of the edge direction*. Had it required directed trees, t_β would not qualify, since none of its nodes can reach the others through unidirectional paths.

The above definition allows arbitrary seed sets, in particular, an S_i can be **N**, the set of all graph nodes. We adjust Def. 2.8 to allow a connecting tree to have any number of nodes *from those seed sets equal to* **N** (otherwise, only 1-node trees would appear in results).

Difference wrt path-based semantics Consider a simple CTP $g' = (v_1, v_2, \underline{v_3})$ and two seed sets S_1, S_2. $g'(S_1, S_2)$ *may differ from* the set of all paths between an S_1 node and an S_2 node: for instance, a path going from $s_1 \in S_1$ *through* $s'_1 \in S_1$ to $s_2 \in S_2$ cannot appear in $g'(S_1, S_2)$, because of our minimality condition (*ii*), requiring *direct* connections between seeds from different sets. Further, consider a CTP $g'' = (v_1, v_2, v_3, \underline{v_4})$ and some seed sets S_1, S_2, S_3. One may try to compute $g''(S_1, S_2, S_3)$ by a three-way join of the paths from a common root node r, to a node from S_1, one from S_2 and one from S_3; we call this approach **path stitching**. The results may differ even more: (*i*) for each tree of n nodes that appears in $g''(S_1, S_2, S_3)$, the three-way join produces n results, that need deduplication; (*ii*) if a path from r to s_1 has common nodes or even common edges with a path from r to s_2 and/or the one from r to s_3, the join of these paths is *not a tree*, thus it cannot appear in a CTP result. This is why in this work, we compute CTP results directly (not via stitching).

Note that a CTP can have a very large number of results, as illustrated by the graph in Figure 2. A CTP $(1, N+1, \underline{v_3})$, asking for all the connections between the end nodes, has 2^N solutions, or $2^{|E|/2}$, which grows exponentially in $|E|$, the number of graph edges. This is why **complete CTP result computation may be unfeasible** in some cases, and we will include in our language **CTP filters** for limiting the CTP result computation effort.

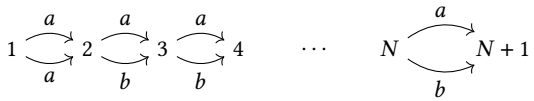


Figure 2: Sample "chain" graph.

We call *simple variable* in a query a variable that does not occur in the last position in a CTP. For a core query Q, we define:

Definition 2.9 (Simple embedding)**.** A simple embedding of Q in **G** is a function ϕ mapping each simple variable into a **G** node or edge, such that:

(1) The restriction of ϕ to each BGP pattern b of Q is an embedding of b into **G** (Def. 2.7);
(2) For each query CTP of the form $g = (g_1, \ldots, g_m, \underline{v_{m+1}})$, such that the simple variable in the predicate g_i, for $1 \leq i \leq m$, is v_i, ϕ maps each v_i into a **G** node satisfying g_i.

Definition 2.10 (Core query result)**.** Let Q be a core query having the head variables $u_1, \ldots, u_n$, and the simple variables $v_1, \ldots, v_p$. Let Φ be the set of all $(\phi(v_1), \ldots, \phi(v_p))$ tuples for any simple embedding ϕ of Q in **G**. For each CTP g^j in Q of the form $(g_1, \ldots, g_m, \underline{v_{m+1}})$, let v_i^j be the simple variable in g_i, $1 \leq i \leq m$. We define the i-th seed set of g^j, denoted S_i^j, as $\pi_{v_i^j}(\Phi)$, that is: all the nodes to which v_i^j is bound in Φ. The result of Q is:

$$Q(\mathbf{G}) = \pi_{u_1,\ldots,u_n}(\Phi \bowtie g^1(S_1^1, \ldots, S_{m_1}^1) \bowtie \ldots \bowtie g^l(S_1^l, \ldots, S_{m_l}^l))$$

where $g^1, \ldots, g^l$ are the CTPs of Q, having respectively m_j simple variables, $1 \leq j \leq l$, $g^j(S_1^j, \ldots, S_{m_j}^j)$ is the set-based CTP result of g^j (Def. 2.8) on its seed sets derived from Φ, and $\bowtie$ denotes the natural join on all the simple variables.

CTP filters A set of orthogonal language extensions, which allow to filter (restrict) set based CTP results, are also provided.

The keyword UNI after a CTP indicates that only *unidirectional* trees are sought, that is: a tree t, as in Def. 2.8, must have a *root* node, from which a *directed path* goes to each seed node in t.

Adding LABEL and a set of labels $\{l_1, l_2, \ldots, l_k\}$ after a CTP indicates that the edges in any result of that CTP must have labels from the given set.

Adding MAX n after a CTP indicates that only trees of at most n edges are sought.

A **score function** σ can be used to assign to each tree in a CTP result a real number $\sigma(t)$ (the higher, the better). Specifying (for a given CTP or for the whole query) SCORE σ [TOP k] means that the results of each CTPs must be scored using σ, and the scores included in the query result. The optional TOP k allows to restrict the CTP result to those having the k-highest σ scores.

Finally, a practical way to limit the evaluation of a CTP (recall the example on Figure 2) is to specify a timeout T (maximum allowed evaluation time); for simplicity, we consider the same T is allotted to each CTP in a query.

Definition 2.11 (Query)**.** A query consists of a core query, together with 0 or more filters for each CTP.

The semantics of a query is easily derived from that of a core query (Def. 2.10), by filtering set-based CTP results accordingly.

3 QUERY EVALUATION STRATEGY

An EQL query consists of a set of BGPs and a set of CTPs. Our evaluation strategy consists of the following steps:
(A) Evaluate each BGP b_i, that is, compute all embeddings of its variables, and materialize them in a table B_i.
(B) For each CTP g^j of the query, of the form $(g^j_1, \ldots, g^j_{m_j}, \underline{v^j_{m_j+1}})$:

(1) For $1 \leq i \leq m_j$, where v^j_i is the variable in g^j_i, compute the seed set S^j_i as follows.
   - If v^j_i appears also in one of the B_i, take S^j_i to be $\pi_{v^j_i}(B_i)$ (all the nodes to which v^j_i has been bound). Further, if g^j_i is not an empty predicate, restrict S^j_i to only those nodes that also satisfy g^j_i.
   - Otherwise, we obtain S^j_i by restricting **N** (the graph's nodes set) to those that match g^j_i.

(2) Compute $F_j(g^j(S^j_1, \ldots, S^j_{m_j}))$, where $F_j(\cdot)$ applies all the CTP filters that may be attached to g^j. In practice, we actually *push the filters in the CTP evaluation*. Thus, we use the notation $g^j(S^j_1, \ldots, S^j_{m_j}, F_j)$ to denote the *set-based result of g^j given its seed sets and filters*, and store it in a table CTP_j.

(C) Compute the query result as a projection on the head variables, over the natural join of the B_i and CTP_j tables.

All the above steps but (B) can be implemented by leveraging an existing conjunctive graph query engine. Thus, in the sequel, we focus on efficiently computing set-based CTP results.

4 COMPUTING SET-BASED CTP RESULTS

To compute $g(S_1, \ldots, S_m, F)$, we must find all the minimal subtrees of $\mathbf{G} = (\mathbf{N}, \mathbf{E})$ containing exactly one node (or **seed**) from each S_i, also taking into account the filters F. Since F is optional, we first discuss how to compute CTP results without any filter (Section 4.1 to 4.7), before discussing pushing filters (Section 4.8).

Observation 1. Let us call **leaf** any node in a tree that is adjacent to exactly one edge. It is easy to see that **in each CTP result, every leaf node is a seed**. (Otherwise, the leaf could be removed while still preserving an answer, which contradicts the minimality of the result.) Clearly, the converse does not hold: in a result, some seeds may be internal nodes. We denote by **sat(*t*)** the node sets from which t has a seed.

Observation 2. As stated in Section 2, we may be only computing *partial* CTP results. In such cases, it is reasonable to *return at least the smallest-size results*, given that tree size (smaller is better) is an ingredient of many score functions (see Section 6), and small results are easy to understand. However, we do not assume "smaller is always better": that is for the score function σ to decide. Nor do we require users to specify a maximum result size, which may be hard for them to guess. Rather, we consider algorithms that *find as many results as possible, as fast as possible*, also taking into account the *CTP filters*, which may limit the search.

Seed set size Most of our discussion assumes that no seed set is **N**, and that they all fit easily in memory. We briefly discuss how the contrary situations could be handled, in Section 4.9.

4.1 Simple Breadth-First algorithm (BFT)

The first algorithm we consider finds the *t*ree results in *b*readth-*f*irst fashion, thus we call it BFT. It starts by creating a first generation of trees T_0, containing a one-node tree, denoted Init (n), for each seed node $n \in S_1 \cup \ldots \cup S_m$. Then, from each generation T_i, it builds the trees T_{i+1}, by "growing" each tree t in T_i, successively, with every edge (n, n') adjacent to one of its nodes $n \in t$, such that:

- (Grow1): n' is not already in t, and
- (Grow2): n' is not a seed from a set $S_j \in \text{sat}(t)$.

Condition (Grow1) ensures we only build trees. (Grow2) enforces the CTP result minimality condition (*ii*) (Def. 2.8). As trees grow from their original seed, they can include more seeds. When a tree has a seed from each set, it must be minimized, by removing all edges that do not lead to a seed, before reporting it in the result. For instance, with the seed sets $\{n_2\}$ and $\{n_4\}$ on the graph in Figure 1, starting from n_2, BFT may build $\{e_5, e_4\}$, then $\{e_5, e_4, e_6\}$ before realizing that e_4 is useless, and removing it through minimization. Minimization slows BFT down, as we experimentally show in Section 5.4.1. BFT can build a tree in multiple ways; to avoid duplicate work, any tree built during the search must be stored, and each new tree is checked against this memory of the search.

It is easy to see that **BFT is complete**, i.e., given enough time and memory, it finds all CTP results.

4.2 GAM algorithm

The GAM (Grow and Aggressive Merge) algorithm has been introduced recently [6], reusing some ideas from [16]. Unlike BFT that views a tree as a set of edges, GAM *distinguishes one root node in each tree* it builds. The algorithm uses a *priority queue* where Grow opportunities are inserted, as (tree, edge) pairs such that the tree could grow from its root with that edge.

GAM also starts from the set of Init trees built from the seed sets. Next, it inserts in the priority queue all (t, e) pairs for some Init tree t and edge e adjacent to the root (only node) of t, satisfying the conditions (Grow1) and (Grow2) introduced in Section 4.1. GAM then repeats the following, until no new trees can be built, or a time-out is reached:

(1) (Grow): Pop a highest-priority (t, e) pair from the priority queue, where $e = (t.root, n')$, and build the tree t^i having all edges of t as well as e, and rooted in n'.
(2) (Merge): For any tree t^{ii} already built, such that:
   - (Merge1): t^{ii} has the same root as t^i, and no other node in common with t^i; and
   - (Merge2): $\text{sat}(t^i) \cap \text{sat}(t^{ii}) = \emptyset$,

   take the following steps:
   (a) Create t^{iii}, a tree having the edges of t^i and those of t^{ii}, and the same root as t^i and t^{ii};
   (b) Immediately Merge t^{iii} with all qualifying trees (see conditions Merge1, Merge2), and again merge the resulting trees etc., until no more Merge are possible;
(3) For each tree t^{iv} created via Grow or Merge as above: (*i*) if t^{iv} has a seed from each set, report it as a result; (*ii*) otherwise, push in the priority queue all (t^{iv}, e^{iv}) pairs such that e^{iv} is adjacent to the (only) root node of t^{iv}, satisfying the conditions (Grow1) and (Grow2).

Property 1 (GAM completeness). The GAM algorithm is complete.

Property 2 (GAM result minimality). By construction, each result tree built by GAM is minimal (in the sense of Def. 2.8).

Thus, **GAM does not need to minimize** the results it finds.

Search space exploration order Unlike BFT, GAM does not build trees in the strictly increasing order of their size; MERGE may build quite large trees before some other, smaller trees. The order in which GAM enumerates trees is determined, first, by the priority of the queue which holds (t, e) entries, and second, by the available MERGE opportunities. In this work, **to remain compatible with any score function, we study search algorithms regardless of (orthogonally to) the search order.**

Like BFT, GAM may also build a tree in multiple ways. Formally:

Definition 4.1 (Tree with provenance). A tree with provenance (or provenance, in short) is a formula of one of the forms shown below, together with one node called the *provenance root*:

(1) INIT (n) where n is a seed; the root of such a provenance is n itself;
(2) GROW (t, e) where t is a provenance, its root is n_0, e is an edge going from n_0 to n_1 and n_1 does not appear in t; in this case, n_1 is the root of the GROW provenance;
(3) MERGE (t_1, t_2), where t_1 and t_2 are provenances, rooted in n_1=n_2; in this case, n_1 is the root of the MERGE provenance.

We call **rooted tree** a set of edges that, together, form a tree, together with one distinguished root node. GAM may build several provenances for the same rooted tree, e.g., MERGE (MERGE (t_1, t_2),t_3) and MERGE (t_2,MERGE (t_1, t_3)), for some trees t_1, t_2, t_3. The interest of a tree as part of a possible result does not depend on its provenance. Therefore, **GAM discards all but the first provenance built for a given rooted tree**.

4.3 BFT variants with MERGE

The MERGE operation can also be injected in the BFT algorithm to allow it to build some larger trees before all the smaller trees have been enumerated. We study two variants: BFT-M *m*erges each new tree resulting from GROW with all its compatible partners (Step (2a) in Section 4.2), but does not apply MERGE on top of these MERGE results; in contrast, BFT-AM applies both Step (2a) and Step (2b) to *a*ggressively *m*erge. BFT-M and BFT-AM are obviously complete. Like BFT, they still need to minimize a potential result before reporting it. This is because BFT algorithms *grow trees from any of their nodes*, thus may add edges on one side of one seed node, which later turn to be useless. GAM avoids this by growing only from the root.

4.4 Edge set pruning and ESP algorithm

GAM may build several rooted trees for the same set of edges. For example, on the graph in Figure 3 with the seeds $\{B\}$, $\{C\}$, **denoting a rooted tree by its edges and underlining the root**, successive GROW from B lead to B-3-C, successive GROW from C lead to B-3-C, and MERGE of two GROW provenances yields B-3-C. However, the root is meaningless in a CTP result, which is simply a set of edges. We introduce:

Definition 4.2 (Edge set). An edge set is a set of edges that, together, form a tree such that at most 1 leaf is not a seed.

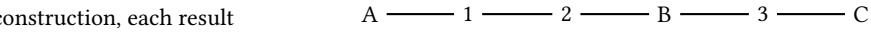


Figure 3: ESP incompleteness example.

A result is a particular case of edge set, where all leaves are seeds (recall Observation 1).

As GAM builds several rooted trees for an edge set, it *repeats some effort*: we only need to find each result once. This leads to the following pruning idea:

Definition 4.3 (Edge-set pruning (ESP)). The ESP pruning technique during GAM consists of discarding any provenance t_1 whose edge set is non-empty, such that another provenance t_0, corresponding to the same edge set, had been created previously.

We will call ESP, in short, the GAM algorithm (Section 4.2) enhanced with ESP. As we will show, **ESP significantly speeds up GAM execution**. However, **ESP compromises completeness** for some graphs, seed sets, *and execution orders*. That is: *depending on the order in which various trees are built*, the first (and only, due to ESP) provenance for a given edge set may prevent the algorithm from finding some results.

For instance, consider the graph in Figure 3, and the seed sets $S_1 = \{A\}, S_2 = \{B\}, S_3 = \{C\}$. A possible execution of GAM is:

(1) Initial trees: A, B, C.
(2) A set of GROW lead to these trees: A–1, B–2, B–3, C–3.
(3) B-3 and C-3 merge into B-3-C.
(4) GROW on A-1 leads to A-1-2, which immediately merges with B-2, forming A-1-2-B.
(5) After this point:
    - If the tree A-1-2-B is built, for instance by GROW on A-1-2, ESP discards it since A-1-2-B was found in step (4). Lacking A-1-2-B, we cannot GROW over it to build the result provenance A-1-2-B-3-C. Nor can we build the result provenance MERGE (A-1-2-B, B-3-C).
    - By a similar reasoning, when B-3-C is built, it is discarded by ESP, preventing the construction of of A-1-2-B-3-C.

    Thus, no result is found.

Note that *with a favorable execution order*, the CTP result would be found. For instance, from A, B, C, ESP could build:

(1) Through successive GROW: A–1, A-1-2, A-1-2-B, C-3, C-3-B
(2) Then, MERGE (A-1-2-B, C-3-B) is a provenance for the result.

This raises the question: can we pick a GAM execution order that would ensure completeness, even when using ESP? Intuitively, the order should ensure that *for each result r, there exists a provenance p_r for r which is certainly built*, which requires that *at every sub-expression e of p_r, over an edge set es, the first provenance p_{es} we find for es happens to be rooted in a node that allows to build on e until p_r*. Thus, the decisions made up to building p_{es} would need to have a "look-ahead" knowledge of the *future* of the search, which is clearly not possible. In the above example the "bad" order builds A-1-2-B first, whereas it would be more favorable to build A-1-2-B. However, when exploring these three edges, the future of the exploration is not known; thus, we cannot "pre-determine" the best provenance for es. Recall also from Section 4.2 that different orders may be suited for partial exploration with different score functions. In a conservative way, we consider an algorithm incomplete when *for some "bad" execution order* it may miss results.

We show that ESP finds *some* answers *for any execution order*:

Property 3 (2-seed sets ESP completeness). Let t be a result of a CTP with 2 seed sets. Then, t is guaranteed to be found by ESP.

Here and throughout this paper, *guaranteed to be found*, for a rooted tree or an edge set, means that at least one provenance for it is built; ESP cannot prune the one built first.

For 1 seed set, Property 3 is trivially shown, thus we focus on $m = 2$ (two seed sets). In this case, any result is path of 0 or more edges. We introduce:

Definition 4.4 ((n, s)-rooted path). Given a CTP and its seed sets $S_1, S_2, \ldots, S_m$, an (n, s)-*rooted path* is a rooted path from a seed s to a root node n, such that the only seed in the path is s.

Lemma 4.1. *Any (n, s)-rooted path is guaranteed to be found by GAM with ESP.*

Proof. We prove this by exhibiting a provenance for it. First, for each seed $s \in S_1 \cup \ldots \cup S_m$, Init (s) is guaranteed to be built. ESP pruning does not apply. Then, any provenance applying only Grow steps on an Init provenance, is guaranteed to be built by GAM. Such a provenance is not pruned by ESP, because it is the *only* provenance that could lead to its edge set. Thus, successive Grow on top of any seed s is guaranteed to build up to n, leading to the (n, s)-rooted path. □

Based on the above lemma, we prove Property 3:

Proof. If the result t is a node ($s_1 = s_2$), the property is trivial. If the result is a path of 1 edge, there are two provenances of the form Grow (Init); the first is already a result. Now, assume t has at least two edges. For any internal node n in t, the (n, s_i)-rooted paths from both the (seed) leaves s_1, s_2 of t are guaranteed to be found, by Lemma 4.1. Then, one of two cases may occur: **(1)** For some internal node n_0, both rooted paths (n_0, s_1) and (n_0, s_2) are created *before* a sequence of Grow gets from Init (s_1) to s_2, and *before* the opposite sequence of Grow is built from Init (s_2), to s_1. Without loss of generality, let n_0 denote the *first* internal node for which these two rooted paths are created. Immediately, Merge on these creates a provenance of t. By the way we chose n_0, this is the first provenance for this edge set, thus not pruned. **(2)** On the contrary, assume that successive Grow get from one end of the path to another, *before* two rooted paths meet in any internal node. Assume without loss of generality that Grow (Grow (…Init (s_1)…)) is the first one to reach s_2. Again, by design, this is the first provenance for t, thus not pruned. □

CTP with two seed sets (path queries) are frequent in practice; on these, GAM [6] and ESP are comparable, and we experimentally show the latter is much more efficient. Next, we add more algorithmic refinements to significantly extend our completeness guarantees.

4.5 MoESP algorithm

We now introduce an algorithmic variant called *Merge-oriented ESP*, or MoESP, which finds many (but not all) CTP results for arbitrary numbers of seed sets.

MoESP works like ESP, but it creates more trees. Specifically, whenever Grow or Merge produces a provenance t *having strictly more seeds than any of its (one or two) children*, the algorithm builds from t all the so-called **MoESP trees** t' such that:

- t' has the same edges (and nodes) as t, but
- t' is rooted in a seed node, distinct from the root of t.

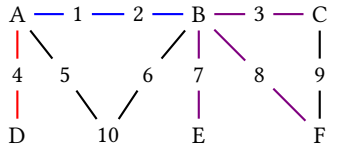


Figure 4: Sample graph for MoESP discussion.

The provenance of any such t' is denoted Mo (t, r) where Mo is special symbol and r is the root of t'. Within MoESP, **Merge is allowed on MoESP trees, but not Grow**. More generally, Grow is disabled on any tree whose provenance includes Mo.

Clearly, MoESP builds a strict superset of the rooted trees created by ESP (thus, it finds all results of ESP). It also finds the result in Figure 3. Namely, after creating A, B, C:

(1) Grow leads to the trees: A–1, B–2, B–3, C–3.
(2) B-3 and C-3 merge into B-3-C. MoESP trees are added at this point: B-3-C and B-3-C.
(3) Grow on A-1 leads to A-1-2, which merges with B-2, forming A-1-2-B. Similarly, A-1-2-B and A-1-2-B are added.
(4) A-1-2-B merges with B-3-C, leading to the result.

We now generalize the example by establishing completeness guarantees for MoESP.

Definition 4.5 (Simple and p-simple edge set). A simple edge set is an edge set (Def. 4.2) where each leaf is a seed and no internal (non-leaf) node is a seed. A simple edge set is p-simple, for some integer p, if its number of leaves is at most p.

For instance, consider the sample graph in Figure 4, and the 6 seed sets $\{A\}$, $\{B\}$, $\{C\}$, $\{D\}$, $\{E\}$, $\{F\}$. The edge set A-4-D, shown in red, is 2-simple, and so are: A-1-2-B, shown in blue; B-8-F, etc.

Definition 4.6 (Simple tree decomposition of a solution). Let t be a CTP result. A simple tree decomposition of t, denoted $\theta(t)$, is a set of simple edge sets which (i) are a partition of the edges of t and (ii) may share (leaf) nodes with each other.

For instance, in Figure 4, the red, blue, and violet edges, together, form a result for the 6-seed sets CTP. A simple tree decomposition of this solution is: {A-4-D, A-1-2-B, B-7-E, B-8-F, B-3-C}. It is easy to see that a tree t has a unique simple tree decomposition $\theta(t)$.

Definition 4.7 (p-piecewise simple solution). A result t is p-piecewise simple ($p$ps, in short), for some integer p, if every edge set in the simple tree decomposition $\theta(t)$ is p-simple (Def. 4.5).

The sample result above in Figure 4 is 2ps, since its simple tree decomposition only contains 2-simple edge sets. The following important MoESP property guarantees it is found:

Property 4 (MoESP finds 2-piecewise simple solutions). For any number of seed sets m, MoESP is guaranteed to find any 2-piecewise simple result.

Proof. Let t be a 2-piecewise simple solution and $\theta(t) = \{t_1, \ldots, t_r\}$ be its simple tree decomposition. It is easy to see that each t_i, $1 \leq i \leq r$, is a path of the form $n_1^i, \ldots, n_m^i$ such that n_1^i and n_m^i

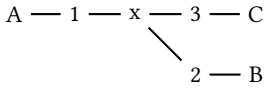


Figure 5: MoESP incompleteness example.

are seeds, while no other intermediary node is a seed. Lemma 4.1, which still holds for MoESP, guarantees that rooted paths are built starting from both n_1^i and n_m^i. As soon as these paths meet, a tree over the edges of t_i is created, then thanks to MoESP, one tree rooted in n_1^i and another rooted in n_m^i, over the edge set of t_i, are created. Because $\theta(t)$ is a simple tree decomposition of t, if $r = 1$, the property is proved. If $r > 1$, each seed-rooted tree based on the edge set of a t_i has its root in common with at least another seed-rooted tree over another edge set(s) from $\theta(t)$. Therefore, aggressive MERGE ensures that they are eventually all merged, leading to one provenance for t. □

For a CTP with any number m of seed sets, a *path result* is one in which no node has more than two adjacent edges. In a path result, seed and non-seed nodes alternate, with the two ends of the paths being seeds. Thus, any path result is 2ps. It follows then, as a direct consequence of Property 4:

Property 5 (MoESP finds all path results). For any CTP, MoESP finds all the path results.

However, outside 2ps results, MoESP may still fail. For instance, consider the graph in Figure 5, and the seed sets $\{A\}, \{B\}, \{C\}$. The only result here is 3-simple. A possible MoESP execution order is:

(1) Starting from A, B, C, GROW produces A–1, B–2, C–3;
(2) B–2–x, followed by B–2–x–3, which merges with C-3 into B–2–x–3–C, leading also to B–2–x–3–C and B–2–x–3–C.
(3) B-2-x-1 which merges with A-1, leading to B-2-x-1-A and similar trees rooted in B and A.
(4) GROW produces A-1-x. ESP discards the MERGE of A-1-x with B–2–x, due to the rooted tree built at step (3), over the same set of edges.
(5) A–1–x–3 is built, then MERGE with C-3 creates A–1–x–3–C, and similar trees rooted in A and C.
(6) GROW produces C-3-x. ESP discards the merges of C-3-x with A-1-x due to the 3-rooted tree built at step (5) and with B–2–x due to the 3-rooted tree built at step (2).
(7) At this point, we have trees with two seeds, rooted in 1, 3, A, B and C. GROW on any of them is impossible, because they already contain all the edges adjacent to their roots. There are no MERGE possibilities on their roots, either. Thus, the search fails to find a result.

At steps (4) and (6), ESP is "short-sighted": it prevents the construction of some trees, necessary for finding the result. Next, we present another optimization which prevents such errors.

4.6 LESP algorithm

The Limited Edge-Set Pruning (LESP), in short, works like ESP (Section 4.4), but it *limits* edge-set pruning, as follows.

- We assign to each node n, and maintain throughout LESP execution, a *seed signature* ss_n, indicating the seed sets S_i, $1 \leq i \leq m$, such that a (n, s_i)-rooted path (Def. 4.4) has been built from a seed $s_i \in S_i$, to n, since execution started. For any seed $s \in S_i$, the signature ss_s is initialized to $0 \ldots 1 \ldots 0$ (a single 1 in the i-th position). For a non-seed n, initially ss_n=0; the i-th bit is set to 1 when node n is reached by the first rooted path from a seed in S_i.
- Prevent ESP from discarding a MERGE tree rooted in n such that: (i) $\sum(ss_n) \geq 3$, that is, there are at least 3 bits set to 1 in the signature ss_n; and (ii) n has at least 3 adjacent edges in **G**.

Intuitively, the condition on ss_n *encourages merging on nodes already well-connected to seeds*. We denote by d_n the number of **G** edges adjacent to n; it can be computed and stored before evaluating any query. The condition on d_n focuses the "protection against ESP" to MERGE trees *rooted in nodes where such protection is likely to be most useful*: specifically, those where 3 or more rooted paths can meet (see Lemma 4.2 below). GROW and MERGE apply on trees "spared" in this way with no restriction.

Clearly, LESP creates all the trees built by ESP, and may create more. In particular, reconsider the graph in Figure 5, the associated seed sets, and the execution steps we traced in Section 4.5. At step (2), ss_x is initialized with 010 (there is a path from B to x). At step (4), when A-1-x is built, ss_x becomes 110; since $\sum(ss_x) = 2$, the tree A-1-x-2-B is pruned. However, at step (6), when C-3-x is built, ss_x becomes 111, which, together with $d_x = 3$, spares its MERGE result A-1-x-3-C (despite the presence of several trees with the same edges). In turn, this merges immediately with B-2-x into a result.

We formalize the guarantees of LESP as follows.

Definition 4.8 ((u, n) rooted merge). For an integer $u \geq 3$ and non-seed node n, the (u, n) rooted merge is the rooted tree resulting from merging a set of u (n, s_i) rooted paths, for some seeds $s_1, \ldots, s_u$.

It follows from the (MERGE2) pre-condition (Section 4.2) that in an (u, n) rooted merge, each s_i belongs to a different seed set. Further, it follows from the definition of an (n, s_i)-rooted path, that in a (u, n) rooted merge, all seeds are on leaves. In other words, a (u, n) rooted merge is a u-simple edge set.

LEMMA 4.2. *Any* $(3, n)$ *rooted merge is guaranteed to be found by LESP.*

PROOF. For any non-seed node n, Lemma 4.1 (which also holds for LESP) ensures that any (n, s_i)-rooted path is found. As soon as the third one is built, $\sum(ss_n)$ becomes 3. This, and the hypothesis $d_n \geq 3$, ensure that the MERGE of the three is not pruned. □

Property 6. For any integer $u \geq 3$ and non-seed node n, any (u, n) rooted merge is guaranteed to be found by LESP.

PROOF. For $u = 3$ this is established by Lemma 4.2. Once the first $(3, n)$ rooted merge has been built and kept, this ensures both that $d_n \geq 3$ and $\sum(ss_n) \geq 3$. Then, whenever a new (n, s_i) rooted path, satisfying the MERGE pre-conditions, is built, it is aggressively merged with the first $(3, n)$ rooted path, and the result is protected from pruning by LESP's special provision. The same holds during all subsequent merges with other (n, s_j) rooted paths. □

For 4 or more seed sets, LESP may miss results that are not (u, n) rooted merges. For instance, consider the following order of execution for $S = (\{A\}, \{B\}, \{C\}, \{D\})$ on the graph in Figure 6:

(1) From A, B, C, D, GROW builds: A–1, B–2, C-3, D-4.
(2) GROW builds B–2-1 which merges with A-1 into A-1–2-B.

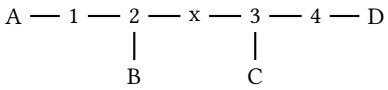


Figure 6: LESP incompleteness example with 4 seed sets.

(3) Grow builds C-3-4 which merges with D-4 into C-3-4-D.
(4) Grow builds: A-1-2; B–2-x which cannot merge with B-2 as A-1–2-B exists, and $\sum(ss_2) = 2$; D-4-3 which cannot merge with C-3 as C-3-4-D exists, and $\sum(ss_3) = 2$.
(5) C-3-x merges with B–2-x to build B–2-x-3-C.
(6) C-3-x-2 merges with: A-1-2, leading to C-3-x-2-1-A; and B–2, leading to C-3-x-2-B.
(7) Similarly, B-2-x-3, aggressively merges with C-3, leading to B-2-x-3-C, and D-4-3, leading to B-2-x-3-4-D.
(8) Progressing similarly, we can only merge at most 3 rooted paths, in nodes 2, x or 3. We cannot merge with a path leading to the 4th seed, because the trees with the edge sets A-1-2-B and C-3-4-D, built at (2), (3) above, are not rooted in 2 nor 3, respectively, and these are the only nodes satisfying the LESP condition that "spares" some Merge trees.

4.7 MoLESP algorithm

Our last algorithm, called MoLESP, is a GAM variant with ESP and *both* the modifications of MoESP (which injects more trees) and LESP (which avoids ESP pruning for some Merge trees). Clearly, MoLESP finds all the trees found by MoESP and LESP. Further:

Property 7 (MoLESP finds all 3ps results). MoLESP is guaranteed to find all the 3-piecewise simple results.

Proof. Let t be a 3ps result. If t was 2ps, MoESP finds it (Property 4), thus MoLESP also does.

Now consider that $\theta(t)$ has some 3-simple edge sets that are not 2-simple (thus, $m \geq 3$). We show that for any 3-simple edge set in $\theta(t)$, one provenance is built. Let t^3 be such an edge set: its three leaves, denoted n_1, n_2, n_3, are seeds, and no internal node is a seed. Let c denote the central node in t^3 (connected to n_1, n_2, n_3 by pairwise disjoint paths). t^3 is a $(3, c)$ rooted merge (recall Def. 4.8) and one provenance for it is built (Lemma 4.2).

The rest of the proof follows the idea in the proof of Property 4. The MoESP aspect of MoLESP guarantees that for each edge set in $\theta(t)$, one tree rooted in each seed is built and not pruned; eventually, aggressive Merge of these trees builds a provenance for t. □

As an important consequence:

Property 8. MoLESP is complete for $m \leq 3$ seed sets.

Proof. Consider the possible result shapes: (*i*) a single node $s_1 = s_2 = s_3$: no ESP applies, thus it is found; (*ii*) a path going from $s_1 = s_2$ to s_3; such a result is 2-simple; (*iii*) a path going from s_1 to s_2 and then to s_3, for some pairwise distinct s_1, s_2, s_3; such a result is 2ps; (*iv*) a tree with three distinct leaves s_1, s_2, s_3, which is 3-simple. In cases (*ii*), (*iii*), (*iv*), Property 7 ensures the result is found. □

Our strongest completeness result is:

Property 9 (Restricted MoLESP completeness). For any CTP of $m \geq 1$ seeds, MoLESP finds any result t, such that: each edge set $es \in \theta(t)$ is a (u, n)-rooted merge (Def. 4.8), for some integer $1 \leq u \leq m$ and non-seed node n in es.

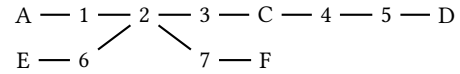


Figure 7: MoLESP completeness example.

Algorithm 1: MoLESP(graph **G**, seed sets $(S_1 \ldots, S_m)$)

```
   Output: Set of results, Res
1  Priority queue PrioQ ← new priority queue;
2  History Hist ← new set of edge sets;
3  foreach S_i, 1 ≤ i ≤ m do
4  |  foreach n_i^j ∈ S_i do
5  |  |  t_i^j ← Init (n_i^j); processTree(t_i^j);
6  |  end
7  end
8  while PrioQ is not empty do
9  |  (t, e) ← poll(PrioQ); t' ← Grow (t, e);
10 |  Update ss_{root(t')}; processTree(t');
11 end
```

Algorithm 2: Procedure processTree(provenance t)

```
1  if isNew(t) then
2  |  Add t to Hist ;
3  |  if isResult(t) then
4  |  |  Add t to Res;
5  |  end
6  |  else
7  |  |  recordForMerging(t);
8  |  |  if t is not a MoESP tree then
9  |  |  |  for edge e ∈ adjacentEdges(t.root) do
10 |  |  |  |  if hasNotBeenInQueue(t, e) then
11 |  |  |  |  |  Add (t, e) to PrioQ;
12 |  |  |  |  end
13 |  |  |  end
14 |  |  end
15 |  end
16 end
```

Algorithm 3: Procedure recordForMerging(tree t)

```
1  Add t to TreesRootedIn[t.root];
2  for n ∈ (nodes(t) ∩ ∪_i(S_i)) do
3  |  Copy t into a new tree t', rooted at n, with provenance
   |    Mo(t, n);
4  |  Add t' to TreesRootedIn[n];
5  |  MergeAll(t');
6  end
```

Proof. Let t be a result, and assume it is v-piecewise simple, for some integer v. If $v \in \{2, 3\}$, Property 7 ensures MoLESP finds it.

On the contrary, assume $v \geq 4$ and let $t^4 \in \theta(t)$ be a (v, n)-rooted merge for some non-seed node n, thus, also v-simple. Property 6, which also holds during MoLESP, guarantees that one provenance for t^4 is built. The end of our proof leverages the MoESP aspect of the algorithm: for each such edge set in $\theta(t)$, one tree rooted in each seed is built and not pruned; eventually, aggressive Merge of these trees builds a provenance for t. □

For example, in Figure 7, with the six seeds A to F, the result is guaranteed to be found by MoLESP. Depending on the exploration order, MoESP and LESP may not find it.

MoLESP algorithm Algorithms 1 to 5, together, implement MoLESP. They share a set of global variables whose names start with an uppercase letter: **Res**, **PrioQ**, **Hist** (the search history), and **TreesRootedIn** (to store the trees by their roots); the latter is needed to find Merge candidates fast. Variables with lowercase names are local to each algorithm. processTree feeds the priority queue with (tree, edge) pairs at line 10. recordForMerging injects the extra MoESP trees

Algorithm 4: Procedure isNew(tree t)

```
1  if t ∉ Hist then
2  |  return true;
3  end
4  if Σ(ss_{t.root}) ≥ 3 and d_{t.root} ≥ 3 then
5  |  if t ∉ TreesRootedIn[t.root] then
6  |  |  return true;
7  |  end
8  end
9  return false;
```

Algorithm 5: Procedure MergeAll(tree t)

```
1  toBeMerged ← {t};
2  while toBeMerged ≠ ∅ do
3  |  currentTrees ← toBeMerged; toBeMerged ← ∅;
4  |  for t' ∈ currentTrees do
5  |  |  mergePartners ← TreesRootedIn[t'.root];
6  |  |  for t_p ∈ mergePartners do
7  |  |  |  if sat(t')∩sat(t_p)= ∅ and t' ∩ t_p = {t'.root}
   |  |  |  |  then
8  |  |  |  |  t'' ← Merge (t', t_p);
9  |  |  |  |  if isNew(t'') then
10 |  |  |  |  |  Add t'' to toBeMerged;
11 |  |  |  |  |  processTree(t'');
12 |  |  |  |  end
13 |  |  |  end
14 |  |  end
15 |  end
16 end
```

(Section 4.5) at lines 2 to 4. isNew implements limited edge-set pruning based on the history, and the two conditions that can "spare" a tree from pruning (Section 4.6). mergeAll implements aggressive merging; by calling processTree on each new Merge result, through recordForMerging, the result is available in the future iterations of mergeAll, thus ensuring all the desired Merge.

4.8 CTP evaluation in the presence of filters

We now briefly explain how various CTP filters (Section 2) can be inserted within the above algorithms. UNI-directional search is enforced by adding pre-conditions to Grow and Merge, to ensure we only create the desired provenances. LABEL $\{l_1, l_2, \ldots, l_k\}$ is enforced by restricting the Grow edges to only those carrying one of these labels; in GAM and its variants, we only add in the queue (line 10 in processTree), (tree, edge) pairs where the edge has an allowed label. MAX n prevents Grow and Merge from creating a tree of more than n edges. timeout T is checked after each newly found rooted tree and within each algorithm's main loop.

For SCORE σ [TOP k], the simplest implementation calls σ on each new result; a vast majority of the proposed score functions can score each result independently. If the score of a result can only be computed once *all* the results are found, e.g. [37, 38], the results need to be accumulated. For **any given score** σ, a smarter implementation may *favor (with guarantees, or just heuristically) the early production of higher-score results, by appropriately chosing the priority queue order*; this allows search to finish faster. **Any** order can be chosen in conjunction with MoLESP, since its completeness guarantees are independent of the exploration order.

4.9 Handling very large seed sets

Our CTP evaluation algorithms build Init trees for each seed. This has two risks: (*i*) when one or more seed sets are **N** (all graph nodes), exploring them all may be unfeasible; (*ii*) one or more seed sets may be subsets of **N**, yet still much larger, e.g., one or more orders of magnitude, than the other seed sets. To handle (*i*), assuming other seed sets are smaller, we only start exploring (Init, Grow etc.) from the other seed sets, and simplify accordingly the algorithms, since any encountered node is acceptable as a match for the **N** seed set(s). To handle (*ii*), borrowing ideas from prior work [26], we use *multiple priority queues*, one for each subset of the seed sets, and Grow at any point *from the queue having the fewest (tree, edge) pairs*. Thus, exploration initially focuses on the neighborhood of the smaller seed sets, and hopefully encounters Init trees from the large seed sets, leading to results.

5 EXPERIMENTAL EVALUATION

We compare CTP evaluation algorithms, then consider systems capable, to some extent, to evaluate the language we introduced.

5.1 Software and hardware setup

We implemented a parser and a query compiler for our language (Section 2) as an extension of SPARQL, and all the CTP evaluation algorithms from Section 4, in Java 11. Our graphs are stored in a simple table graph(id,source, edgeLabel, target) within PostgreSQL 12.4; unless otherwise specified, we delegate to Postgres the BGP evaluation, and joining their results with CTP ones (Section 3). When comparing CTP evaluation algorithms with in-memory competitors, we load the graph in memory prior to evaluating CTPs.

We executed our experiments on a server equipped with 2x10-core Intel Xeon E5-2640 CPUs @ 2.4GHz, with 128-GB DRAM. Every execution point is averaged over 3 executions.

5.2 Baselines

CTP evaluation (keyword search) algorithms Our focus is on algorithms that search for connecting trees (*i*) traversing edges in both directions, (*ii*) orthogonally wrt the score function, (*iii*) exhaustively, at least up to m=3 seed sets, (*iv*) capable of returning as many solutions as requested, if given enough time and memory, and (*v*) applicable to arbitrary graphs, i.e., not requiring a regular graph structure. In the literature, only the GAM algorithm [6] (Section 4.2) fits the bill. The BFT, BFT-M, BFT-AM algorithms (Section 4.1 and 4.3) also satisfy these conditions, and are thus natural comparison baselines; like virtually all algorithms from the literature, they start from the seeds and move gradually away looking for results.

QGSTP [39] and LANCET [40] are the most recent GSTP approximation algorithms, for specific cost functions based on node and edge (LANCET) weights. LANCET relies on DPBF [16] to find an initial result, which it then improves. Since QGSTP has shown strong advantage over DPBF [39], we select QGSTP as a baseline. QGSTP runs in polynomial time in the size of the graph, and by design, returns only *one* result; we used the authors' code.

Graph query engines Our first two baselines only support *checking, but not returning* unbounded-length, unidirectional paths whose edge labels match a regular expression that users *must* provide, that is: one cannot ask for "any path". Specifically, we use **Virtuoso** OpenSource v7.2.6 to evaluate SPARQL 1.1 queries that come as close as possible to the semantics of our language. Internally, Virtuoso translates an incoming SPARQL query into an SQL dialect[1]

[1] Accessible using the built-in function `sparql_to_sql_text()`.

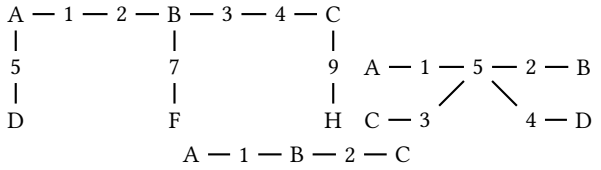


Figure 8: Synthetic graphs: Comb(3, 1, 2, 3) at the top left, Star(4, 2) at the top right, and Line(3, 1) at the bottom.

before executing it. Our second baseline, named **Virtuoso-SQL**, consists of editing these SQL-like queries to remove label constraints and thus query the graph for connectivity between nodes. However, Virtuoso's SQL dialect prevented us from *returning* the nodes and edge labels along the found paths (whereas standard recursive SQL allows it).

Our next three baselines support checking *and returning* paths. **JEDI** [2] returns all the data paths matching a SPARQL property path; we use the authors' code. **Neo4j** supports Cypher queries asking for all directed or undirected paths between two sets of nodes. Finally, we used recursive queries in **Postgres** v12.4 to return the label on paths between node pairs.

5.3 Datasets and queries

We experiment with both synthetic and real-world RDF graphs.

To compare CTP evaluation algorithms, we generate three sets of parameterized graphs and associated CTPs (Figure 8). The seeds are labeled $A, B, \ldots, H$, non-seed nodes are labeled 1, 2 etc.; each seed set is of size 1. **Line**(m, nL) contains m seeds, each connected to the next/previous seed by nL intermediary nodes, using sL=nL+1 edges. **Comb**(nA, nS, sL, dBA) consists of a line, from which a lateral segment (called *bristle*) exits each seed. There are nA bristles, each made of nS segments (a segment ends in another seed); each bristle segment has sL triples, and there are dBA nodes in the main line between two successive bristles. The number of seeds is m=nA·(nS+1). **Star**(m, sL) has a central node connected to each of the m seeds by a line of sL edges.

On each Line, Comb, and Star graph, we run a CTP defined by the m seeds, having 1 result. For instance, on the Star in Figure 8, the seed sets are $\{A\}, \{B\}, \{C\}, \{D\}$. On Line and Comb, the result is 2ps (Def. 4.7), while on Star, it is a (u, n) rooted merge (Def. 4.8). Thus, by Property 9, MoLESP is guaranteed to find them. The topology of Line graphs minimizes the number of subtrees for a given number of edges and seeds; specifically, there are $O((m{\cdot}nL)^2)$ subtrees, while the number of rooted trees is in $O((m{\cdot}nL)^3)$. On the contrary, the Star topology raises the number of subtrees to $O(2^m{\cdot}sL^2)$, while its number of rooted trees is in $O(2^m{\cdot}sL^3)$. In Comb and Line graphs, MoESP trees (Section 4.5) are part of results.

To study the evaluation of our extended query language, we generate parameterized **Connected Dense Forest (CDF)** graphs (see Figure 9). Each graph contains a *top forest*, and a *bottom forest*; each of these is a set of N_T disjoint, complete binary trees of depth 3. *Links* connect leaves from the top and bottom forests. We generate CDFs for $m{\in}\{2,3\}$: when m=2, chains of triples connect a top leaf to a bottom one; when m=3, a Y-shaped connection goes from a top-forest leaf, to two bottom-forest ones. A CDF graph contains N_L links, each made of S_L triples. Only top leaves that are targets of "c" edges can participate to links, and we concentrate the links on 50% of them (the others have no links). When m=2, only 50% of the

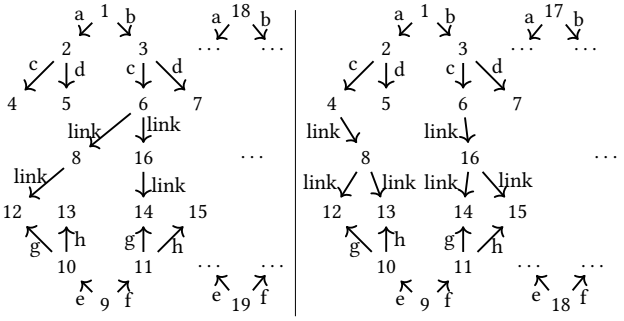


Figure 9: CDF graphs generated with m=2, S_L=2 (left), and with m=3, S_L = 3 (right).

bottom forest leaves that are targets of "g" edges can participate; when m=3, 50% of all the bottom forest leaf can participate. The links are uniformly distributed across the eligible leaves. A CDF has $12{\cdot}N_T{+}N_L{\cdot}S_L$ edges; it has $14{\cdot}N_T{+}N_L{\cdot}(S_L{-}1)$ nodes if m=2, and $14{\cdot}N_T{+}N_L{\cdot}S_L$ if m=3.

On CDF graphs with m=2, we run the query (v,tl,$\underline{l}$) :- (x,"c",tl), (v,"g",bl), (bl,tl,$\underline{l}$) whose two BGPs bind tl, respectively, bl to leaves from the top and bottom forest, while its CTP asks for all the paths between each pair of such leaves. On graphs with m=3, we run (v,tl,$\underline{l}$):- (x, "c", tl), (v,"g",bl_1), (v,"h",bl_2),(tl, bl_1, bl_2, $\underline{l}$), requiring connecting trees between tl, bl_1 and bl_2. Each CDF query has N_L answers, one for each link.

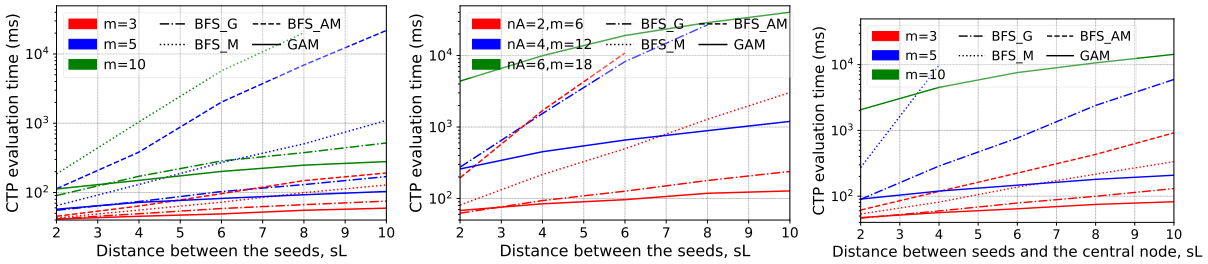
Real-world graphs To compare with JEDI [2] and QGSTP [39], we reused their datasets (a 6M triples subset of YAGO3, and a 18M triples subset of DBPedia), as well as their queries.

5.4 CTP evaluation algorithms

5.4.1 Complete (baseline) algorithms. We start by comparing the algorithms without any pruning: BFT (Section 4.1), GAM (Section 4.2), and the BFT variants BFT-M and BFT-AM (Section 4.2), on synthetic Line, Comb and Star graphs of increasing size. We used a timeout T of 10 minutes. **In all experiments with GAM and all its variants, our exploration order (queue priority) favors the smallest trees, and breaks ties arbitrarily**. Figure 10 depicts the algorithm running time; the color indicates the number of seed sets (3, 5 or 10), while the line pattern indicates the algorithm. *Missing points (or curves) denote algorithms that did not complete by the timeout.* Note the logarithmic y axes.

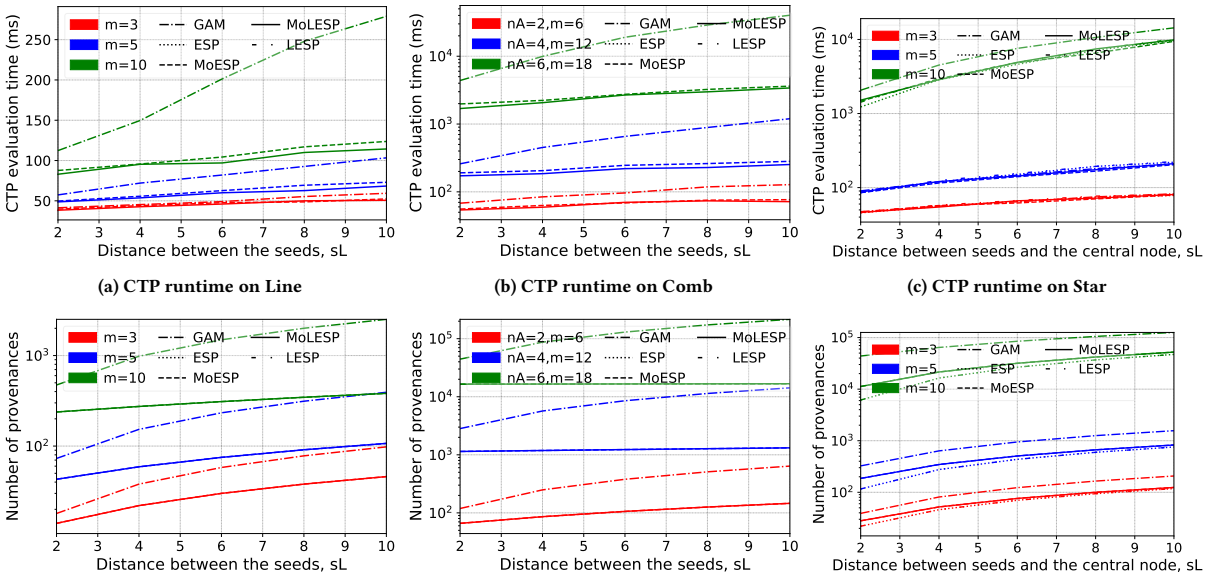
Across these plots, **BFT-M performs worse than BFT-AM**. On Line graphs, the difference is a factor 2× for m = 3 and up to 100× for m = 10. On the Comb and Star graphs, BFT-M times out on the larger graphs and queries. **BFT-AM takes even more than BFT-M**, by a factor of 15×, thus more executions timed out. **GAM is much faster** and completes execution in all cases. The reason, as explained in Section 4.1, is that breadth-first algorithms waste effort by minimizing results, and may find a tree in even more different ways than GAM, since they grow from any node. Thus, *we exclude breadth-first algorithms from the subsequent comparisons.*

5.4.2 GAM algorithm variants. On the same graphs, we compare GAM (Section 4.2), ESP (Section 4.4), MoESP (Section 4.5), LESP (Section 4.6) and MoLESP (Section 4.7) with the same timeout. Figure 11 shows the algorithm running time as well as the number of provenances they built. In all graphs but Figure 11a, the y axis is



(a) CTP runtime on Line (b) CTP runtime on Comb (c) CTP runtime on Star

Figure 10: Comparison of complete CTP evaluation baselines.



(a) CTP runtime on Line (b) CTP runtime on Comb (c) CTP runtime on Star

(d) Number of provenances on Line (e) Number of provenances on Comb (f) Number of provenances on Star

Figure 11: Graphs for GAM variants on synthetic benchmarks.

logarithmic. On Line and Comb graphs, **ESP and LESP failed to find results** due to edge set pruning, as explained in Section 4.4, thus the corresponding curves are missing. MoESP and MoLESP build the same number of provenances on Line and Comb graphs.

The plots show, first, that edge set pruning significantly reduces the running time: **MoLESP is faster than GAM** by a factor ranging from 1.3× (Line graphs) to 15× (Comb graphs, nA=6, m=18). Second, on the Star graphs, where the limited edge-set pruning (Section 4.6) applies, the performance difference between MoESP and MoLESP is small. This shows that **the extra cost incurred by LESP and MoLESP, which limit or compensate for edge-set pruning** (by injecting more trees), **is worth paying for the completeness guarantees** of MoLESP. Overall, the algorithm running times closely track the numbers of built provenances, further highlighting the interest of controlling the latter through pruning.

5.4.3 Comparison with QGSTP on real-world data. We now compare the winner of the above comparisons, namely MoLESP, with QGSTP [39] on the 18M edges DBPedia dataset and 312 CTPs used in their evaluation. Among these, 83 CTPs (respectively, 98, 85, 38, 8) have 2 (respectively, 3, 4, 5, 6) seed sets. To align with QGSTP, we added a UNI filter (unidirectional exploration only), and LIMIT 1 to stop after the first result. Each QGSTP returned result is such that Property 9 ensures MoLESP finds it. Figure 12 shows the average runtimes grouped by m. GAM is faster than QGSTP for $m \leq 5$, but timed-out for the 8 CTPs with m=6. MoLESP is about 6-7× faster than QGSTP for all m values, and scales well as m increases. Thus, **MoLESP is competitive also on large real-world graphs and queries**.

5.5 Extended query evaluation

5.5.1 Synthetic queries on CDF benchmark. We now compare our EQL query evaluation system with the graph query baselines, on our CDF graphs (Section 5.3) generated with $m \in \{2,3\}$, $S_L \in \{3,6\}$, 18K to 2.4M edges, leading to 2K up to 200K results (N_L), respectively. We used T=15 minutes. As explained in Section 2, the paths returned by the baselines, which we "stitch" for m=3, semantically differ from CTP results; the baselines' reported time *do not include the time to minimize nor deduplicate their results.*

For m=2, Figure 13 shows that all systems scale linearly in the input size (note the logarithmic time axis). *For each system, the lower curve is on graphs with S_L=3, while the upper curve is on graphs with S_L=6 (these graphs are larger, thus curves go farther at right). All missing points correspond to time-out.* JEDI succeeded only on the

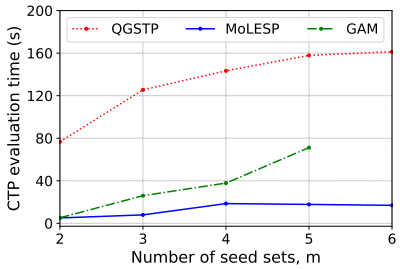


Figure 12: GAM and MoLESP vs. QGSTP [39] on DBPedia.

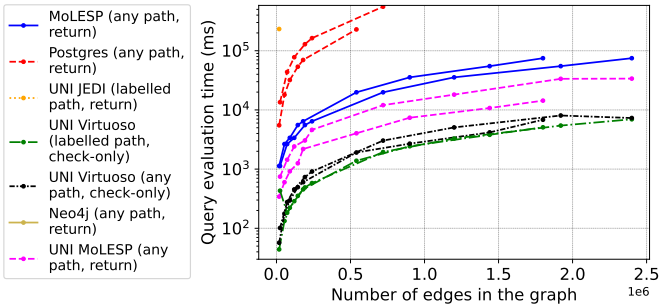


Figure 13: CDF benchmark performance for m=2, S_L∈{3,6}.

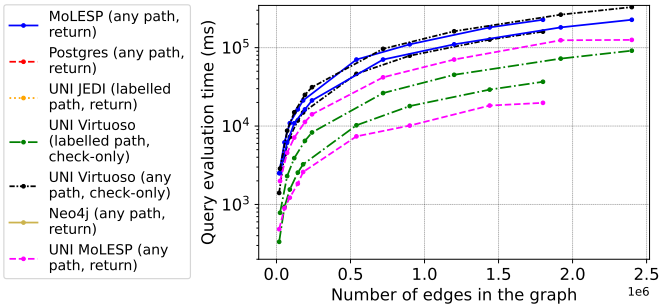


Figure 14: CDF benchmark performance for m=3, S_L∈{3,6}.

smallest graph, Neo4j timed-out on all. Virtuoso-SPARQL is the fastest, closely followed by Virtuoso-SQL; they are both *unidirectional, require the edge labels*, and *do not return paths*. Unidirectional MoLESP, which we included to compare with unidirectional baselines, is slower by approximately 3× only. JEDI is slower than MoLESP by 10^2× on the smallest graph, and timed-out on the others. Postgres is faster than JEDI, yet at least 10× slower than MoLESP. **MoLESP is the only feasible bidirectional algorithm**; it runs in under 2 minutes on the largest graph with 2.4M edges.

Figure 14 shows similar results for m=3. Postgres timed-out in all cases. Virtuoso-SPARQL is 7× faster than Virtuoso-SQL; both return *non-minimal, duplicate results*. UNI-MoLESP outperforms every system, while *also returning connecting trees*. Note that the *bidirectional* MoLESP found about 7× more results than the N_L expected ones, by also connecting bottom leaves *without a common parent* through their grandparent node; these results are filtered by the join between the BGPs and the CTP (Section 3). Despite the much larger search space due to bidirectionality, MoLESP scales well with the size of the graph.

5.5.2 Comparison with JEDI on real-world data. JEDI [2] used a set of (unidirectional, label-constrained) SPARQL 1.1 queries over YAGO3. Table 1 shows the queries' characteristics. We compare MoLESP similarly constrained (UNI and LABEL), on these queries,

Query	JEDI	MoLESP	Virtuoso	Neo4j
J_1: 3 BGPs, 2 CTPs	3.9	1.9	0.2	TimeOut
J_2: 2 BGPs, 1 CTP, large seed set	0.9	1	OOM	TimeOut
J_3: 1 CTP, N seed set	0.75	2.3	OOM	1.27

Table 1: Query evaluation times (seconds) on YAGO3 dataset.

with JEDI, Virtuoso and Neo4j (Postgres timed-out on all). Query J_2 has one very large seed set, while query J_3 has a N seed set. *On queries J_2 and J_3, MoLESP timed out.* Thus, we applied the optimizations described in Section 4.9, which enabled it to perform as shown. Virtuoso-SPARQL completed query J_1, then ran out of memory. Compared with JEDI, our query evaluation engine is 2× faster on J_1, close on J_2, and around 3× slower on J_3. MoLESP took around 30% of the total time, the rest being spent by Postgres in the BGP evaluation and final joins. This shows that **the optimizations described in Section 4.9 make MoLESP robust also to large seed sets.**

6 RELATED WORK AND PERSPECTIVES

We focused on extending a **graph query language**, such as SPARQL [13], Cypher [35] or GraphQL [18], with *connecting tree patterns* (CTPs) that they currently do not support (our requirement (**R1**) from Section 1). Specifically, SPARQL 1.1 property paths (*i*) allow to *check* that some paths connect two nodes, not to *return* the path(s); (*ii*) do not allow searching for *arbitrary* paths (users have to specify a regular expression); (*iii*) are restricted to *unidirectional* paths only. Some PG query languages such as Neo4j's Cypher lift these restrictions, however, its implementation does not scale (Section 5.5.1) [11]. RPQProv [15] uses recursive SQL to return path labels; JEDI [2, 3] builds over SPARQL 1.1 by returning all unidirectional paths. Many works focus on finding label-constrained paths between nodes [7, 8, 17, 19, 20, 28, 34, 36, 42, 43, 45], typically by using precomputed indexes or sketches. In our CTP evaluation algorithm, an index could be integrated by "reading from it" paths (or subtrees) on which to GROW and MERGE. *Our CTPs extend finding paths, to finding trees that connect an arbitrary number of seed sets (m≥3), traversing edges in any direction* by default; we guarantee completeness for m≤3 and finding a large set of results for arbitrary m. As we explained (Section 2), path stitching leads to different results, which may require deduplication and minimization.

The CTP evaluation problem is directly related to **keyword search in (semi-)structured data**, addressed in many algorithms, some of which are surveyed in [12, 44]. These prior studies differ from ours as follows: (*i*) [4, 14, 21, 23–25, 27, 32, 33, 41] are schema-dependent; (*ii*) [9, 29, 41] assume available a compact summary of the graph; (*iii*) [16, 22, 26, 31] depend heavily on their score functions for pruning the search, particularly to approximate the best result [16, 31] or return only top-k results [10, 22, 30, 32, 46]; (*iv*) [1, 4, 10, 22, 24] are only unidirectional. For these reasons, they fail to meet our requirements (**R2**) to (**R5**) as outlined in Section 1.

The Java-based GAM algorithm used in this work [6] was sped up by up to 100× in a multi-threaded, C++ version [5]. MoLESP brings new, orthogonal, optimizations, and novel guarantees.

Our future work includes developing adaptive EQL optimization and execution strategies and applying it to graph exploration for investigative journalism.

Acknowledgments. This work is funded by AI Chair SourcesSay project (ANR-20-CHIA-0015-01) grant.

REFERENCES

[1] B. Aditya, Gaurav Bhalotia, Soumen Chakrabarti, Arvind Hulgeri, Charuta Nakhe, Parag, and S. Sudarshan. 2002. BANKS: Browsing and Keyword Searching in Relational Databases. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*. 1083–1086. https://doi.org/10.1016/B978-155860869-6/50114-1

[2] Christian Aebeloe, Gabriela Montoya, Vinay Setty, and Katja Hose. 2018. Discovering Diversified Paths in Knowledge Bases. *Proc. VLDB Endow.* 11, 12 (2018), 2002–2005. https://doi.org/10.14778/3229863.3236245 Code available at: http://qweb.cs.aau.dk/jedi/.

[3] Christian Aebeloe, Vinay Setty, Gabriela Montoya, and Katja Hose. 2018. Top-K Diversification for Path Queries in Knowledge Graphs. In *Proceedings of the ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks co-located with 17th International Semantic Web Conference (ISWC 2018), Monterey, USA, October 8th - to - 12th, 2018 (CEUR Workshop Proceedings)*, Marieke van Erp, Medha Atre, Vanessa López, Kavitha Srinivas, and Carolina Fortuna (Eds.), Vol. 2180. CEUR-WS.org. http://ceur-ws.org/Vol-2180/paper-01.pdf

[4] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. 2002. DBXplorer: A System for Keyword-Based Search over Relational Databases. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, Rakesh Agrawal and Klaus R. Dittrich (Eds.). IEEE Computer Society, 5–16. https://doi.org/10.1109/ICDE.2002.994693

[5] Angelos-Christos Anadiotis, Oana Balalau, Théo Bouganim, Francesco Chimienti, Helena Galhardas, Mhd Yamen Haddad, Stéphane Horel, Ioana Manolescu, and Youssr Youssef. 2021. Empowering Investigative Journalism with Graph-based Heterogeneous Data Management. *Bulletin of the Technical Committee on Data Engineering* (Sept. 2021). https://hal.archives-ouvertes.fr/hal-03337650

[6] Angelos-Christos G. Anadiotis, Oana Balalau, Catarina Conceição, Helena Galhardas, Mhd Yamen Haddad, Ioana Manolescu, Tayeb Merabti, and Jingmao You. 2022. Graph integration of structured, semistructured and unstructured data for data journalism. *Inf. Syst.* 104 (2022), 101846. https://doi.org/10.1016/j.is.2021.101846

[7] Kemafor Anyanwu, Angela Maduko, and Amit P. Sheth. 2007. SPARQ2L: towards support for subgraph extraction queries in rdf databases. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*. 797–806. https://doi.org/10.1145/1242572.1242680

[8] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, and Javiel Rojas-Ledesma. 2022. Time- and Space-Efficient Regular Path Queries on Graphs. (2022).

[9] Andrey Balmin, Vagelis Hristidis, and Yannis Papakonstantinou. 2004. ObjectRank: Authority-Based Keyword Search in Databases. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*. 564–575. https://doi.org/10.1016/B978-012088469-8.50051-6

[10] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. 2002. Keyword Searching and Browsing in Databases using BANKS. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*. 431–440. https://doi.org/10.1109/ICDE.2002.994756

[11] Andrew Bowman. 2022. Tuning Cypher queries by understanding cardinality. (2022). https://neo4j.com/developer/kb/understanding-cypher-cardinality/#_distinct_nodes_from_variable_length_paths

[12] Joel Coffman and Alfred C. Weaver. 2014. An Empirical Performance Evaluation of Relational Keyword Search Techniques. *IEEE Trans. Knowl. Data Eng.* 26, 1 (2014), 30–42. https://doi.org/10.1109/TKDE.2012.228

[13] WWW Consortium. 2013. SPARQL 1.1. (2013). https://www.w3.org/TR/sparql11-overview/

[14] Pericles de Oliveira, Altigran S. da Silva, Edleno Silva de Moura, and Rosiane Rodrigues. 2018. Match-Based Candidate Network Generation for Keyword Queries over Relational Databases. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. 1344–1347. https://doi.org/10.1109/ICDE.2018.00146

[15] Saumen C. Dey, Víctor Cuevas-Vicenttín, Sven Köhler, Eric Gribkoff, Michael Wang, and Bertram Ludäscher. 2013. On implementing provenance-aware regular path queries with relational query engines. In *Joint 2013 EDBT/ICDT Conferences, EDBT/ICDT '13, Genoa, Italy, March 22, 2013, Workshop Proceedings*, Giovanna Guerrini (Ed.). ACM, 214–223. https://doi.org/10.1145/2457317.2457353

[16] Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. 2007. Finding Top-k Min-Cost Connected Trees in Databases. (2007), 836–845. https://doi.org/10.1109/ICDE.2007.367929

[17] George H. L. Fletcher, Jeroen Peters, and Alexandra Poulovassilis. 2016. Efficient regular path query evaluation using path indexes. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016*, Evaggelia Pitoura, Sofian Maabout, Georgia Koutrika, Amélie Marian, Letizia Tanca, Ioana Manolescu, and Kostas Stefanidis (Eds.). OpenProceedings.org, 636–639. https://doi.org/10.5441/002/edbt.2016.67

[18] The GraphQL Foundation. 2022. GraphQL. (2022). https://graphql.org/

[19] Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert. 2013. Sparqling kleene: fast property paths in RDF-3X. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-located with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*. 14. https://doi.org/10.1145/2484425.2484443

[20] Andrey Gubichev and Thomas Neumann. 2011. Path Query Processing on Very Large RDF Graphs. In *Proceedings of the 14th International Workshop on the Web and Databases 2011, WebDB 2011, Athens, Greece, June 12, 2011*. http://webdb2011.rutgers.edu/papers/Paper21/pathwebdb.pdf

[21] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. 2003. XRANK: Ranked Keyword Search over XML Documents. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*. 16–27. https://doi.org/10.1145/872757.872762

[22] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. 2007. BLINKS: ranked keyword searches on graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*. 305–316. https://doi.org/10.1145/1247480.1247516

[23] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. 2003. Efficient IR-Style Keyword Search over Relational Databases. In *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*. 850–861. https://doi.org/10.1016/B978-012722442-8/50080-X

[24] Vagelis Hristidis and Yannis Papakonstantinou. 2002. DISCOVER: Keyword Search in Relational Databases. In *VLDB*. http://www.vldb.org/conf/2002/S19P02.pdf

[25] Vagelis Hristidis, Yannis Papakonstantinou, and Andrey Balmin. 2003. Keyword Proximity Search on XML Graphs. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*. 367–378. https://doi.org/10.1109/ICDE.2003.1260806

[26] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. 2005. Bidirectional Expansion For Keyword Search on Graph Databases. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. 505–516. http://www.vldb.org/archives/website/2005/program/paper/wed/p505-kacholia.pdf

[27] Gjergji Kasneci, Maya Ramanath, Mauro Sozio, Fabian M. Suchanek, and Gerhard Weikum. 2009. STAR: Steiner-Tree Approximation in Relationship Graphs. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*. 868–879. https://doi.org/10.1109/ICDE.2009.64

[28] Jochem Kuijpers, George Fletcher, Tobias Lindaaker, and Nikolay Yakovets. 2021. Path Indexing in the Cypher Query Pipeline. In *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*. 582–587. https://doi.org/10.5441/002/edbt.2021.68

[29] Wangchao Le, Feifei Li, Anastasios Kementsietsidis, and Songyun Duan. 2014. Scalable Keyword Search on Large RDF Data. *IEEE Trans. Knowl. Data Eng.* 26, 11 (2014), 2774–2788. https://doi.org/10.1109/TKDE.2014.2302294

[30] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. 2008. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*. 903–914. https://doi.org/10.1145/1376616.1376706

[31] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2016. Efficient and Progressive Group Steiner Tree Search. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 91–106. https://doi.org/10.1145/2882903.2915217

[32] Yi Luo, Xuemin Lin, Wei Wang, and Xiaofang Zhou. 2007. Spark: top-k keyword query in relational databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*. 115–126. https://doi.org/10.1145/1247480.1247495

[33] Yi Luo, Wei Wang, Xuemin Lin, Xiaofang Zhou, Jianmin Wang, and Keqiu Li. 2011. SPARK2: Top-k Keyword Query in Relational Databases. *IEEE Trans. Knowl. Data Eng.* 23, 12 (2011), 1763–1780. https://doi.org/10.1109/TKDE.2011.60

[34] Inju Na, Ilyeop Yi, Kyu-Young Whang, Yang-Sae Moon, and Soon J. Hyun. 2022. Regular Path Query Evaluation Sharing a Reduced Transitive Closure Based on Graph Reduction. (2022).

[35] Inc. Neo4j. 2022. Cypher Query Language. (2022). https://neo4j.com/developer/cypher/

[36] You Peng, Xuemin Lin, Ying Zhang, Wenjie Zhang, and Lu Qin. 2022. Answering reachability and K-reach queries on large graphs with label constraints. *VLDB J.* 31, 1 (2022), 101–127. https://doi.org/10.1007/s00778-021-00695-0

[37] Vinay M. S. and Jayant R. Haritsa. 2019. Root Rank: A Relational Operator for KWS Result Ranking. In *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data, COMAD/CODS 2019, Kolkata, India, January 3-5, 2019*. 103–111. https://doi.org/10.1145/3297001.3297014

[38] Vinay M. S. and Jayant R. Haritsa. 2020. Operator implementation of Result Set Dependent KWS scoring functions. *Inf. Syst.* 89 (2020), 101465. https://doi.org/10.1016/j.is.2019.101465

[39] Yuxuan Shi, Gong Cheng, Trung-Kien Tran, Evgeny Kharlamov, and Yulin Shen. 2021. Efficient Computation of Semantically Cohesive Subgraphs for Keyword-Based Knowledge Graph Exploration. In *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, and Leila Zia (Eds.). ACM / IW3C2, 1410–1421. https://doi.org/10.1145/3442381.3449900 Code available at: https://github.com/nju-websoft/QGSTP.

[40] Yahui Sun, Xiaokui Xiao, Bin Cui, Saman K. Halgamuge, Theodoros Lappas, and Jun Luo. 2021. Finding Group Steiner Trees in Graphs with both Vertex and Edge Weights. *Proc. VLDB Endow.* 14, 7 (2021), 1137–1149. https://doi.org/10.14778/3450980.3450982

[41] Thanh Tran, Haofen Wang, Sebastian Rudolph, and Philipp Cimiano. 2009. Top-k Exploration of Query Candidates for Efficient Keyword Search on Graph-Shaped (RDF) Data. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China.* 405–416. https://doi.org/10.1109/ICDE.2009.119

[42] Lucien D. J. Valstar, George H. L. Fletcher, and Yuichi Yoshida. 2017. Landmark Indexing for Evaluation of Label-Constrained Reachability Queries. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017.* 345–358. https://doi.org/10.1145/3035918.3035955

[43] Sarisht Wadhwa, Anagh Prasad, Sayan Ranu, Amitabha Bagchi, and Srikanta Bedathur. 2019. Efficiently Answering Regular Simple Path Queries on Large Labeled Networks. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* 1463–1480. https://doi.org/10.1145/3299869.3319882

[44] Haixun Wang and Charu C. Aggarwal. 2010. A Survey of Algorithms for Keyword Search on Graph Data. In *Managing and Mining Graph Data*, Charu C. Aggarwal and Haixun Wang (Eds.). Advances in Database Systems, Vol. 40. Springer, 249–273. https://doi.org/10.1007/978-1-4419-6045-0_8

[45] 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, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016.* 1875–1889. https://doi.org/10.1145/2882903.2882944

[46] Yueji Yang, Divyakant Agrawal, H. V. Jagadish, Anthony K. H. Tung, and Shuang Wu. 2019. An Efficient Parallel Keyword Search Engine on Knowledge Graphs. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019.* 338–349. https://doi.org/10.1109/ICDE.2019.00038