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*—When graph database users explore unfamiliar graphs, potentially with heterogeneous structure, users 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*.**

In this work, we formally show how to integrate *connecting tree patterns* (CTPs, in short) with a graph query language such as GPML [1], SPARQL or Cypher, leading to *Extended Queries* (or EQs, in short). We then study a set of algorithms for evaluating CTPs; we generalize prior keyword search work to be complete, most importantly by (i) considering bidirectional edge traversal, (ii) allowing users to select *any* score function for ranking CTP results and (iii) returning *all* results. To cope with very large search spaces, we propose efficient pruning techniques and formally establish a large set of cases where our best algorithm, MOLESP, is complete even with pruning. Our experiments validate the performance of our algorithms on many synthetic and real-world workloads.

I. Introduction

Graph databases are increasingly adopted in many applications, e.g., social network analysis, scientific data exploration, the financial industry, etc. [2]. To query RDF graphs, the W3C's standard SPARQL [3] query language is the best known. For property graphs (PG), the International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) are developing GQL, with the graph pattern matching sub-language (GPML) [1] at its core.

An interesting but challenging graph query language feature is *reachability*: a SPARQL 1.1 query can check, e.g., if there are some bank transfer paths, along which Mr. Shady eventually deposits funds into a given bank ABC. However, SPARQL 1.1 does not allow returning the matching paths to users. In contrast, a GPML query may also return the paths between two given sets of nodes. This is useful in investigative journalism applications [4], in the fight against money laundering, etc.

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

The above query has two parts: one *structured*, requiring that Mr. Shady be a person, ABC be the name of a bank and DEF that of a country, and an *unstructured* one, searching for connections between the nodes designated by the structured one. The unstructured part recalls *keyword search in (relational, RDF or XML) databases*: users specify m keywords, and request trees (or small graphs) connecting tuples (or nodes) from the database, such that a tuple attribute value (or node label) matches each keyword. Representative keyword search works include, e.g., [5]–[14]. Finding such trees is closely related to the Group Steiner Tree Problem (GSTP), which, given m node sets, asks for *the top-score*, e.g., fewest-edges, tree connecting one node from each set; the problem is NP-hard. To cope with the high complexity, existing keyword search algorithms: (i) consider a fixed score 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 with no guarantees, but which perform well in some cases.

Requirements From our collaborations with investigative journalists [4], [15], we identified some requirements. 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 function* used to 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. In our example, 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 should not 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 must be found* for several reasons: (i) continuity with the semantics of graph query languages, that also return all results (unless users explicitly restrict them); (ii) to be able to score the answers with the chosen function (recall R2). We call a tree search algorithm supporting (R2)-(R4) **complete**. For practical reasons, given the complexity, which is further exacerbated by (R3), we need to be able to limit the search by a time and/or

*Work done while at Ecole Polytechnique.

space budget. Finally, (**R5**) *our 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 integrate Connecting Tree Patterns (CTPs, in short) with conjunctive graph pattern queries at the core of both SPARQL and GPML, leading to *Extended Queries (EQs, in short)*. A CTP allows searching for trees that connect m groups of nodes, for $m \geq 2$; they can be freely joined with graph patterns. This addresses requirements (R1), (R2), and also (R3), since our CTP semantics returns trees regardless of the edge directions (Sec. II).

(2) For *complete CTP evaluation*, we study a set of baseline algorithms, and show that their performance suffers due to repeated (wasted) work and/or the need to minimize the trees they find; the GAM [15] algorithm is complete and 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 (R4) and contributes to (R5) (Sec. IV).

(3) We experimentally show that: (i) baseline algorithms inspired from breadth-first search are unfeasible even for small graphs; (ii) MoLESP strongly outperforms the prior complete GAM algorithm [15]; (iii) integrating MoLESP with a simple conjunctive graph query engine allows to efficiently evaluate extended queries (Sec. V).

II. Extended Queries (EQs)

We present connection search as a novel query primitive, and show how it integrates in GPML [1] graph querying.

Definition II.1 (Graph)**.** A graph is defined as a tuple $\mathbf{G} = (\mathbf{N}, \mathbf{E}, \rho, \lambda, \pi)$, where $\mathbf{N}$ consists of a finite set of node identifiers, $\mathbf{E}$ is a finite set of edge identifiers, ρ: $\mathbf{E} \rightarrow (\mathbf{N} \times \mathbf{N})$ is a function mapping edges to ordered pairs of nodes, λ: $(\mathbf{N} \cup \mathbf{E}) \rightarrow 2^{\mathcal{L}}$ is a function mapping nodes and edges to a set of labels, $\mathcal{L}$ and π: $(\mathbf{N} \cup \mathbf{E}) \times \mathcal{P} \rightharpoonup Val$ is a partial function assigning values from Val to node and edge properties ($\mathcal{P}$).

Fig. 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.

GPML graph querying Let $\mathcal{V}$ be a set of variable names, and Ω={=,<,≤} be a set of comparison operators, used to express patterns over nodes or edges. At the core of GPML are **path patterns** (PPs, in short), such as the following:

`MATCH` (v: Alice `WHERE` $v.\tau$=entrepreneur) −[e: citizenOf]→ (w `WHERE` $w.\tau$=country)

The above PP has two node patterns: (v: Alice `WHERE` $v.\tau$=entrepreneur) and (w `WHERE` $w.\tau$=country), specifying the conditions on, respectively, the start and the end points of the edge pattern −[e: citizenOf]→. The source node pattern is satisfied by the node n_3 in our example graph, while the target node pattern is satisfied by the nodes n_8 and n_{10}. The edge pattern is satisfied by the edges e_5, e_6, e_{12}, e_{13}, e_{14} and e_{20} as all of them have the label citizenOf. Together, the PP is satisfied by only the edge $n_3 \xrightarrow{e_{14}} n_8$.

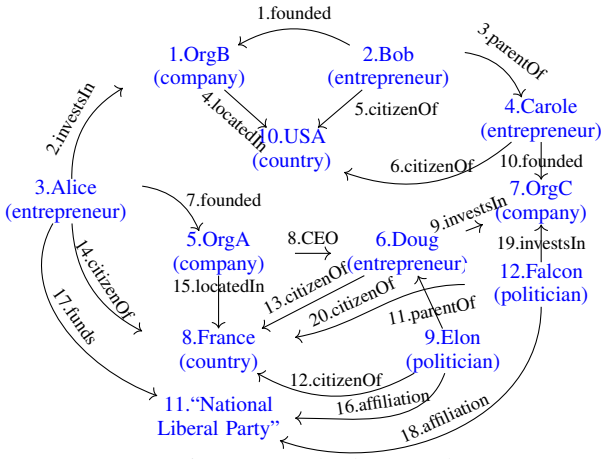


Fig. 1: Sample data graph.

PPs may use *path variables*, which are bound to whole paths. For example, in the PP `MATCH` p = (x) −[y: founded]→ (z) ←[u: investsIn]− (v), one of the bindings is as follows: p matches the path $n_4 \xrightarrow{e_{10}} n_7 \xleftarrow{e_9} n_6$, x matches n_4, y matches e_{10}, z matches n_7, u matches e_9 and v matches n_6.

GPML also supports **regular path queries** (RPs, in short). For example, `MATCH` p = (x) −[y: founded]→∗(z) matches all paths of 0 or more edges labeled founded. Infinitely many paths (even cyclic ones) may match an RP. To ensure finite results, GPML provides *restrictors*, namely: `TRAIL` (no repeated edges), `ACYCLIC` (no repeated nodes) and `SIMPLE` (no repeated nodes except that it could be a loop). If no restrictor is used, GPML mandates limiting RP results through a *selector*, e.g., ANY k for at most k results. A **GPML graph pattern** (GP, in short) is a set of several PPs such that each PP shares a variable with at least one other PP.

GPML generalizes both RDF query languages such as SPARQL 1.1 and property graph query languages such as Cypher; significantly, GPML allows binding variables to paths and thus returning them. However, it does not allow searching for arbitrary connections between *more than two variables*.

Extending GPML: Connecting Tree Patterns To extend GPML's expressive power, we introduce:

Definition II.2 (CT Pattern)**.** A connecting tree pattern (CTP, in short) is a tuple of m+1 distinct variables of the form: $(v_1, v_2, \ldots, v_m, \underline{v_{m+1}})$. Intuitively, CTPs extend the path patterns by enabling connectivity search between more than 2 nodes, as follows. When replacing each v_i, $1 \leq i \leq m$ with a graph node, v_{m+1} is bound to a *subtree* of $\mathbf{G}$, having these nodes as leaves (we formalize this below). To visually distinguish the last variable in a CTP, we always underline it.

For example, consider the CTP $(x, y, z, \underline{w})$; on replacing x with n_3, y with n_9 and z with n_{12}, one subtree bound to

$\underline{w}$ is the tree t_α consisting of the edges $n_3 \xrightarrow{e_{17}} n_{11} \xleftarrow{e_{16}} n_9, n_{11} \xleftarrow{e_{18}} n_{12}$. Tree t_α is one connection between the three nodes, via n_{11}.

We extend GPML to allow CTPs alongside GPs:

Definition II.3 (Extended query)**.** An extended query Q is a set of k GPs, $k \geq 0$, and l CTPs, $l \geq 0$, such that $k+l > 0$, the variables in the GPs are pairwise disjoint, and each underlined (last) variable from a CTP appears exactly once in Q.

Below, we simply use "query" to designate an extended one. A sample query Q_1, of three GPs and one CTP, is:

MATCH
(x WHERE $x.\tau$ = entrepreneur) $-[a$: citizenOf$]\rightarrow$ (b: USA),
(y WHERE $y.\tau$ = entrepreneur) $-[c$: citizenOf$]\rightarrow$ (d: France),
(z WHERE $z.\tau$ = politician) $-[e$: citizenOf$]\rightarrow$ (f: France),
$(x, y, z, \underline{w})$

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.

Definition II.4 (Set-based CTP result)**.** Let $g = (v_1, \ldots, v_m, \underline{v_{m+1}})$ be a CTP pattern and $S_1, \ldots, S_m$ be sets of $\mathbf{G}$ nodes, called **seed sets**. 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 $s_1 \in S_1$, ..., $s_m \in S_m$ and t is a *minimal subtree* (thus, acyclic) of $\mathbf{G}$ 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.

Minimality condition (ii) follows from application requirements, e.g., a minimal connection between a person and a company should not include other companies. If connections between more people and companies are sought, they can be obtained by joining more CTPs and/or GPs.

In Fig. 1, let S_1={n_2, n_4} (US entrepreneurs), S_2={n_3, n_6} (French entrepreneurs), and S_3={n_9, n_{12}} (French politicians). Then, $g^1(S_1, S_2, S_3)$ includes $(n_4, n_6, n_{12}, t_\beta)$ where the tree t_β consists of the edges $n_4 \xrightarrow{e_{10}} n_7 \xleftarrow{e_9} n_6, n_7 \xleftarrow{e_{19}} n_{12}$, also denoted by $\{e_{10}, e_9, e_{19}\}$ for brevity. Another result of this CTP is $(n_2, n_3, n_9, t_\gamma)$, with $t_\gamma = \{e_1, e_2, e_{17}, e_{16}\}$. This result only exists because Def. II.4 allows trees to span over $\mathbf{G}$ edges *regardless of the edge direction*. Had it required directed trees, t_γ 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 $\mathbf{N}$, the set of all graph nodes (Sec. V-E2). We adjust Def. II.4 to allow a connecting tree to have any number of nodes *from those seed sets equal to* $\mathbf{N}$ (otherwise, only 1-node trees would appear in results).

The **semantics** of a PP [1] is a table, associating to each variable v a graph node, edge, or path, which, together, satisfy the PP. The semantics of a GP is the natural join of its PP results, e.g., the variables in the GPs of Q_1 are bound to:

x	y	z	a	b	c	d	e	f
n_4	n_6	n_{12}	e_6	n_{10}	e_{13}	n_8	e_{20}	n_8

GPML	Graph Pattern Matching Language [1]
PP	Path patterns [1]
RP	Regular path query [1]
GP	GPML graph pattern [1]
CTP	Connecting tree pattern (Def. II.2)
EQ	Extended query (Def. II.3)
BFT	Breadth-first tree search algorithm (Sec. IV-A)
GAM	Grow and aggressive merge algorithm (Sec. IV-B)
BFT-M, BFT-AM	Breadth-first tree search variants with MERGE (Sec. IV-C)
ESP	Edge set pruning technique, and the GAM+ESP algorithm (Sec. IV-D)
MoESP	Merge-oriented GAM with ESP (Sec. IV-E)
LESP	GAM with limited edge-set pruning (Sec. IV-F)

TABLE I: Acronyms and notations used in this work

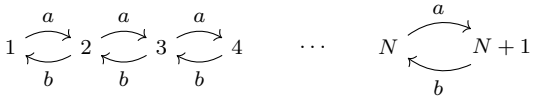


Fig. 2: Sample "chain" graph.

EQ semantics is defined by joining the GP semantics with the CTP set-based semantics, leading to a direct evaluation strategy as follows. Assume that evaluating all the GPs in the EQ, and joining their results, leads to a table of bindings as above, where each node variable v_i is successively bound to a set of nodes; call that set S_i. This provides seed sets for each of the CTPs in the EQ, whose set-based results can thus be computed (Def. II.4). The EQ result is obtained as the natural join of all the GP and CTP results, on their shared variables. In our example, GP semantics provide S_1={n_4}, S_2={n_6} and S_3={n_{12}} to the CTP; CTP evaluation binds $\underline{w}$ to the tree having the edges $n_4 \xrightarrow{e_{10}} n_7 \xleftarrow{e_9} n_6, n_7 \xleftarrow{e_{19}} n_{12}$; the EQ semantics extends the above table with the binding of $\underline{w}$. This evaluation strategy pushes (applies) all possible GP conditions on the CTP node variables prior to CTP evaluation.

Restricting CTP results A CTP can have a very large number of results, as illustrated by the graph in Fig. 2: if v_1 is bound to the node labelled "1" and v_2 to the node labelled "N+1", the CTP $(v_1, v_2, \underline{v_3})$, 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. Observe that if we allowed only unidirectional paths, there would be only $N+1$ results, rooted at each node. This shows that matching CTPs regardless of the edge direction may drastically increase the number of CTP results, and also the number of partial trees to explore before finding the results. In such cases, **complete CTP result computation may be unfeasible**. To control the amount of effort spent evaluating CTPs, similarly to GPML's restrictors, one may specify: that all paths must go from a CTP result root to its leaves (unidirection paths, through the keyword UNI); that only a fixed set of labels (LABEL $a_1, a_2, \ldots$) are allowed on the edges in a CTP result. We also extend the (non-deterministic) GPML selector ANY k to a CTP, specifying

that any k results can be returned; finally, TIMEOUT T stops execution and returns the results found after T milliseconds.

A CTP may be associted a **score function** σ, assigning to each result tree t 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 CTP must be scored using σ, and the scores included in the query result. If TOP k is used, only the k highest-score trees are returned.

Since the GPs can be evaluated by any conjunctive graph query evaluation engine, in the remainder of the paper, we focus on efficient evaluation of set-based CTP results (Def. II.4).

III. CTPs COMPARED TO EXISTING LANGUAGES

We now discuss how close existing graph query languages come to allowing their users to ask for connection between several seed sets, in the spirit of CTPs. First, observe that languages such as SPARQL or CRPQs [16], [17], that do not allow returning a path of unspecified length, cannot be used.

Other languages, e.g., the JEDI extension of SPARQL [18], G-CORE [19], or the GPML pattern matching part of the upcoming ISO/IEC standards SQL/PGQ and GQL [1], allow returning such paths. To see how close this comes to CTPs, assume a GQL query identifies two seed sets S_1, S_2, and asks for all *acyclic* paths between a node from S_1 and one from S_2. Some such paths may not be results of a CTP with the seed sets S_1 and S_2: a path going from $s_1 \in S_1$ *through* $s_1' \in S_1$ to $s_2 \in S_2$ violates minimality condition (ii) from Def. II.4. To ensure this minimality, we should check that each path has exactly one node from S_1 (at one end), but not more, and similarly for S_2. Depending on the language syntax, this may be more or less easy to express.

Next, consider a CTP $g''=(v_1, v_2, v_3, \underline{v_4})$ and *three* seed sets S_1, S_2, S_3 (the discussion is similar for more seed sets). In GPML, we can use three path patterns going from a common variable r, to a node from S_1, one from S_2 and one from S_3, respectively, to obtain triplets of paths joined on a common node matching r. We call this approach for finding connections among the seed sets, **path stitching**. Clearly, each of its results needs to be filtered for minimality, as explained above. However, path stitching results for three or more seed sets differ even more from $g''(S_1, S_2, S_3)$ [20], [21]: (i) for each n-nodes tree in $g''(S_1, S_2, S_3)$, path stitching produces n trees (with the same edges, but different roots); deduplication based on the set of edges is then needed; (ii) if a path from r to s_1 shares nodes or even 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*, but a graph, from which one or several CTP results may be extracted. This is a second, independent reason why trees obtained through path stitching must be globally deduplicated.

As our experiments show (Sec. V-E), even ignoring deduplication and minimization, path stitching is outperformed by our algorithms, where result tree minimality is built-in.

Finally, note that while our CTPs have the core restrictors useful in our application scenarios, languages such as Cypher [22], G-CORE [19], GPML [1] provide more controls over: edge direction, presence of cycles, allowed edge labels, path length etc. One could adapt them to CTPs, with the semantics that they should apply on each path in a CTP result.

IV. COMPUTING SET-BASED CTP RESULTS

We aim to find all the minimal subtrees of $\mathbf{G}$ containing exactly one node (or **seed**) from each S_i. As previously stated, we focus on **complete** algorithms (capable of returning all such trees, orthogonally wrt the score function, and traversing edges in both directions). As we detail in Sec. VI, most prior keyword search algorithms are not complete, and/or require a regular graph structure, which may not be the case. Below, Sec. IV-A to IV-C recall existing complete algorithms. In Sec. IV-D to IV-G, we present our algorithmic contributions, leading to a new, much more efficient algorithm, MOLESP, with interesting completeness guarantees. Sec. IV-H shows how restrictors, selectors and score functions can be injected in this algorithm. Most of our discussion assumes that no seed set is $\mathbf{N}$, and that they all fit in memory. We briefly discuss how to handle the contrary situation in Sec. IV-I.

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. When users limit the search time and/or number of desired results, it is reasonable to *return at least the smallest-size ones*, given that smaller trees are favored by many score functions (see Sec. VI). 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 investigate algorithms that *find as many results as possible, as fast as possible*, while also leveraging restrictors and selectors (when specified) to reduce the evaluation effort.

A. Simple Breadth-First algorithm (BFT)

A natural approach to find trees connecting seed sets is to explore the graph in breadth-first (BF) mode starting from each seed; when search paths starting from one seed of each set meet, a result is found. This approach has been taken by many non-complete algorithms, e.g., [7] [9] [23] and others, which also use score-based heuristics to limit the search. Independent of a score and aiming for completeness, we devise the following simple algorithm, which we call BFT. Start by creating a 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, build 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. II.4). 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 Fig. 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 Sec. V-D1. 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.

B. GAM algorithm

The GAM (Grow and Aggressive Merge) algorithm has been introduced recently [15], reusing some ideas from [9]. Unlike BFT that views a tree as a set of edges, GAM *distinguishes one root node in each tree* it builds. GAM inserts in a *priority queue* GROW opportunities, as (tree, edge) pairs such that the tree could grow from its root with that edge. Any priority can be used in the queue to choose a desired exploration order without affecting the results.

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 Sec. IV-A. 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 queue, where e connects $t.root$ to n', and build the tree t^i having the 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) Build 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} built 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. II.4).

Proof. First, note that the GROW and MERGE conditions ensure that only trees are built, and they have at most one seed from each seed set. Next, we show that *in any tree ever built by GAM, all the leaves (with the possible exception of the root) are seed nodes*. We show that by induction over the tree structure:

- Every initial tree in T_0 consists of one seed.
- Now assume the induction hypothesis is true for a tree t. A GROW (t, e) step adds a new root that is a leaf, and may or may not be a seed; the other leaves of GROW (t, e) are also leaves in t, thus seeds.
- Similarly, assume this holds for two trees t_1, t_2. When t is obtained as MERGE (t_1, t_2), the root of t is by definition not a leaf (it has at least two adjacent edges), while its leaves (those of t_1 and t_2) are seeds, by the induction hypothesis.

Thus, in a GAM tree, all the leaves are seeds; when the root is a leaf, it may or may not be a seed.

We can now finalize proving that GAM builds minimal results, as follows. Step (3)(i) above tests whether each new tree is a result. When this is true of a GROW tree, its root is also a seed, thus Observation 1 holds for GAM results found by GROW. When a MERGE tree is a result, we have shown above that all its leaves are seeds, while the root is by definition ot a leaf. Thus, Observation 1 also holds for GAM results found by MERGE. □

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 large trees before other, smaller trees. The order in which GAM enumerates trees is determined, first, by the priority of the queue with (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.**

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

Definition IV.1 (Tree with provenance). A tree with provenance (or provenance, in short) is a formula of one of the forms below, together with a 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**.

C. 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

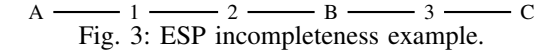


Fig. 3: ESP incompleteness example.

smaller trees have been enumerated. We study two variants: BFT-M *m*erges each new tree resulting from GROW with all partners having only one common node and no common edges (Step (2a) in Sec. IV-B), 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.

D. Edge set pruning and ESP algorithm

GAM may build several rooted trees for the same set of edges. For example, on the graph in Fig. 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, GROW's from C lead to B-3-C, and MERGE of two GROW provenances, B-3 and 3-C yields B-3-C. However, the root is meaningless in a CTP result, which is simply a set of edges. We introduce:

Definition IV.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.

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 IV.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, for the same edge set, had been created previously.

We will call ESP, in short, the GAM algorithm (Sec. IV-B) 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 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, on the graph in Fig. 3, with the seed sets S_1={A}, S_2={B}, S_3={C}, a possible GAM execution 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) By repeated GROW's: A–1, A-1-2, A-1-2-B, C-3, C-3-B
2) 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, a provenance p_r 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 is 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, instead of the favorable 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 Sec. IV-B 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.

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 by ESP.

Here and throughout this paper, *guaranteed*, for a rooted tree or an edge set, means that at least one provenance for it is built; the first is not pruned by ESP.

For 1 seed set, Prop. 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 IV.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 path from a seed s to a root node n, containing only one seed node (s).

Lemma IV.1. *Any (n, s)-rooted path is guaranteed 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. 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 Prop. 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 $\geq$2 edges. For any internal node n in t, the

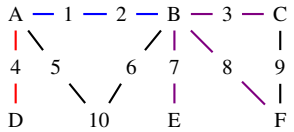


Fig. 4: Sample graph for MoESP discussion.

(n, s_i)-rooted paths from both the (seed) leaves s_1, s_2 of t are guaranteed to be found, by Lemma IV.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 choice of 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. 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 [15] and ESP are comparable, and we experimentally show the latter is much more efficient. Next, refine our algorithms to extend our completeness guarantees.

E. 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. 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.

The provenance of any such t' is denoted MO (t, r) where MO is new 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 as our MoESP completeness guarantees (Prop. 5) do not rely on GROW over such trees.

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 Fig. 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, into A-1-2-B. We similarly get A-1-2-B and A-1-2-B.
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 IV.5 (Simple and p-simple edge set). A simple edge set is an edge set (Def. IV.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 it has at most p leaves.

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

Definition IV.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 Fig. 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}. A tree t has a unique simple tree decomposition $\theta(t)$.

Definition IV.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 is p-simple.

The sample result above in Fig. 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 are seeds, while no other intermediary node is a seed. Lemma IV.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, the two ends of the paths are seeds, while internal nodes may be seeds, or not. Thus, any path result is 2ps. It follows then, as a direct consequence of Prop. 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 Fig. 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;

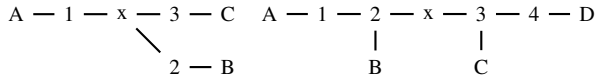


Fig. 5: MoESP(left) and LESP(right) incompleteness example.

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 trees necessary for finding the result. Next, we present another optimization which prevents such errors.

F. LESP *algorithm*

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

- We assign to each node n, and maintain during LESP execution, a *seed signature* ss_n, indicating the seed sets S_i, $1 \leq i \leq m$, such that a rooted path (Def. IV.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 with a 1 at each j such that $s \in S_j$ (s may belong to several seed sets), and 0 in the remaining positions. 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 $\mathbf{G}$.

Intuitively, the condition on ss_n *encourages merging on nodes already well-connected to seeds*. We denote by d_n the number of $\mathbf{G}$ 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 IV.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 Fig. 5, the associated seed sets, and the execution steps we traced in Sec. IV-E. 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.

Algorithm 1: MoLESP($\mathbf{G}$, seed sets $(S_1 \ldots, S_m)$)

```
Output: Set of results, Res
Priority queue PrioQ ← new priority queue;
History Hist ← new set of edge sets;
foreach S_i, 1 ≤ i ≤ m do
    foreach n_i^j ∈ S_i do
        t_i^j ← INIT (n_i^j); PROCESSTREE(t_i^j);
    end
end
while PrioQ is not empty do
    (t, e) ← poll(PrioQ); t' ← GROW (t, e);
    Update ss_root(t'); PROCESSTREE(t');
end
```

Algorithm 2: Procedure PROCESSTREE(provenance t)

```
if ISNEW(t) then
    Add t to Hist ;
    if ISRESULT(t) then
        Add t to Res;
    end
    else
        RECORDFORMERGE(t);
        if t is not a MoESP tree then
            for edge e ∈ adjacentEdges(t.root) do
                if hasNotBeenInQueue(t, e) then
                    Add (t, e) to PrioQ;
                end
            end
        end
    end
end
```

Definition IV.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 (Sec. IV-B) that in an (u, n) rooted merge, each s_i is from 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 leaves. In other words, a (u, n) rooted merge is a u-simple edge set.

Lemma IV.2. *Any* $(3, n)$ *rooted merge is guaranteed to be found by* LESP.

Proof. For any non-seed node n, Lemma IV.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 IV.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 $\geq$4 seed sets, LESP may miss results that are not (u, n) rooted merges. For instance, consider the following execution order for S=($\{A\}, \{B\}, \{C\}, \{D\}$) on the graph in Fig. 5:

1) From A, B, C, D, GROW builds: A–1, B–2, C-3, D-4.
2) GROW builds B–2-1, merging with A-1 into A-1–2-B.
3) GROW builds C-3-4, merging with D-4 into C-3-4-D.
4) GROW builds: A-1-2; B–2-x which cannot merge with B-2 due to A-1–2-B, 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.

G. 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 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. IV.8), thus it is guaranteed (Lemma IV.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), Prop. 7 ensures the result is found. □

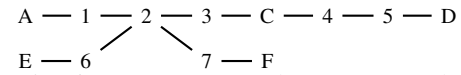


Fig. 6: MOLESP completeness example.

Algorithm 3: Procedure RECORDFORMERGE(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
```

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;
```

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. IV.8), for some integer $1 \leq u \leq m$ and non-seed node n in es.

Proof. Let t be a result, which is v-piecewise simple, for some integer v. If $v \in \{2, 3\}$, MOLESP guarantees it (Prop. 7).

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 t^4. 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 guaranteed; eventually, aggressive MERGE of these trees builds a provenance for t. □

For example, in Fig. 6, with the six seeds A to F, the result is guaranteed 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 global variables whose names start with an uppercase: $\mathbf{Res}, \mathbf{PrioQ}$, $\mathbf{Hist}$ (the search history), and $\mathbf{TreesRootedIn}$ (to store the trees by their roots); the latter is used 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 11. RECORDFORMERGE injects the extra MoESP trees (Sec. IV-E) 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 (Sec. IV-F). MERGEALL implements aggressive merging; by calling PROCESSTREE on each new MERGE result, through RECORDFORMERGE, the result is available in the future iterations of MERGEALL.

H. CTP evaluation with restrictors and score function

We now briefly explain how various restrictors (Sec. II) can be inserted in the above algorithms. UNI-directional search is enforced by adding pre-conditions to GROW and MERGE, to ensure we only create the desired provenances. To enforce

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
```

(LABEL $a_1, a_2, \ldots$), we only add in the queue (line 11 in PROCESSTREE), (tree, edge) pairs where the edge has an allowed label. TIMEOUT T is checked after each newly found rooted tree and within each algorithm's main loop.

For SCORE σ [TOP k], the score σ is evaluated either on each new result, or after all results are found, e.g., for the scores in [24], [25]. For any given score σ, we 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.

I. Handling very large seed sets

Our CTP evaluation algorithms build INIT trees for each seed. This has two risks: (i) when some seed sets are $\mathbf{N}$ (all graph nodes), exploring them may be unfeasible; (ii) one or more seed sets may be very large subsets of $\mathbf{N}$, e.g., orders of magnitude larger than the other seed sets. To handle (i), assuming other seed sets are smaller, we only start exploring (INIT, GROW etc.) from the other (smaller) seed sets; any encountered node is acceptable as a match for the $\mathbf{N}$ seed set(s). To handle (ii), inspired by [7], 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.

V. EXPERIMENTAL EVALUATION

We compare CTP evaluation algorithms, then consider systems capable, to some extent, to evaluate extended queries.

A. Software and hardware setup

We implemented a parser and a query compiler for EQs (Sec. II), and all the CTP evaluation algorithms from Sec. IV, in Java 11. Our graphs are stored graph(id,source, edgeLabel, target) table within PostgreSQL 12.4; unless otherwise specified, we delegate to Postgres the GP evaluation, and joining the result with CTP trees (Sec. II). When comparing CTP evaluation algorithms with in-memory competitors, we also load the graph in memory.

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

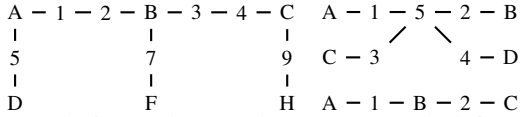


Fig. 7: Synthetic graphs: Comb$(3, 1, 2, 3)$ at the left, Star$(4, 2)$ at the top right, and Line$(3, 1)$ at the bottom right.

B. Baselines

Complete CTP evaluation algorithms We use the algorithms GAM, BFT, BFT-M and BFT-AM as they are the only complete ones, in the sense introduced in Sec. I.

Recent, incomplete CTP algorithms To study how our algorithms compare with incomplete GSTP approximation (graph keyword search) algorithms, we considered the most recent ones: QGSTP [13] and LANCET [14], each leveraging a specific cost function. LANCET relies on DPBF [9] to find an initial result, which it then improves. QGSTP strongly outperforms DPBF [13]; thus we use QGSTP as a baseline. It 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. Specifically, we use **Virtuoso** Open-Source v7.2.6 to evaluate SPARQL 1.1 property path queries. Internally, Virtuoso translates an incoming SPARQL query into an SQL dialect (accessible via the built-in function `sparql_to_sql_text()`) 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 disables *returning* the nodes and edge labels along the paths (whereas standard recursive SQL allows it).

Our next three baselines also allow *returning* paths. **JEDI** [18] returns all 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.

C. Datasets and queries

We experiment on real-life as well as synthetic graphs, aiming to (i) control the parameters impacting the performance of our algorithms, (ii) compare with our baselines and (iii) illustrate actual data journalism application.

To compare CTP evaluation algorithms, we generate three sets of parameterized graphs and associated CTPs (Fig. 7). 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, dB) 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 edges, and there are dB nodes in the main line between two successive bristles. The number of seeds is $m{=}nA{\cdot}(nS{+}1)$. **Star**(m, sL) has a central node connected to each of the m seeds by a line of sL edges.

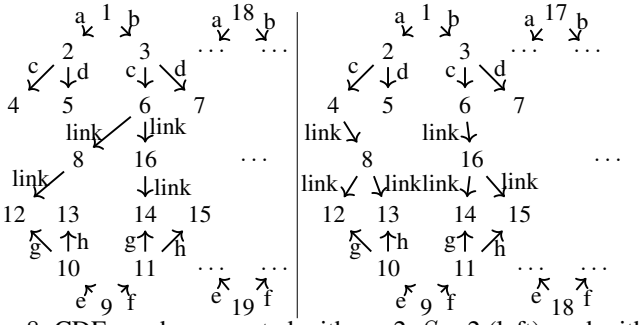


Fig. 8: CDF graphs generated with m=2, S_L=2 (left), and with m=3, $S_L = 3$ (right).

On each Line, Comb, and Star graph, we run a CTP defined by the m seeds, having 1 result. On Line and Comb, the result is 2ps (Def. IV.7), while on Star, it is a (u, n) rooted merge (Def. IV.8). Thus, by Prop. 9, MoLESP guarantees them.

The Line graphs minimize the number of subtrees for a given number of edges and seeds: there are $O((m{\cdot}nL)^2)$ subtrees, and $O((m{\cdot}nL)^3)$ rooted trees. On Star, there are $O(2^m{\cdot}sL^2)$ subtrees, and $O(2^m{\cdot}sL^3)$ rooted trees. In Comb and Line graphs, MoESP trees (Sec. IV-E) are part of results.

We also generated **Barabasi-Albert (BA)** graphs [26] of 200 to 1000 edges. In these graphs, we start with a connected core of 3 nodes; each extra node adds an edge using preferential attachment. Seed nodes are chosen at random. As Table II shows, the random, dense connections of BA graphs yield up to 20.522 results in a 200-edges graph!

To study the evaluation of our extended query language, we generate parameterized **Connected Dense Forest (CDF)** graphs (see Fig. 8). Each graph has 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 edges 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 edges. 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 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, and at most $14{\cdot}N_T{+}N_L{\cdot}S_L$ nodes.

On CDF graphs with m=2, we run the query MATCH (x) $\dashv$[$e1$: c]$\rightarrow$ (tl), (v) $\dashv$[$e2$: g]$\rightarrow$ (bl), $(tl, bl, \underline{l})$ whose two GPs 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 MATCH (x) $\dashv$[e_1: c]$\rightarrow$ (tl), (v) $\dashv$[e_2: g]$\rightarrow$ (bl_1), (v) $\dashv$[e_3: h]$\rightarrow$ (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 results, one for each link. The CDF graphs support both the structured and unstructured parts of our queries, while controlling the number of results.

Real-world graphs To compare with JEDI [18] and QGSTP [13], we reused their datasets (a 6M edges subset of YAGO3, and a 18M edges subset of DBPedia), and queries.

m	$\lvert E\rvert = 200$	$\lvert E\rvert = 400$	$\lvert E\rvert = 600$	$\lvert E\rvert = 800$	$\lvert E\rvert = 1000$
2	38	42	43	43	43
3	833	638	903	778	972
4	20522	16638	15653	21473	17942

TABLE II: Number of results in Barabasi-Albert graphs.

D. CTP evaluation algorithms

1) Complete (baseline) algorithms: We start by comparing the algorithms without any pruning: BFT, GAM, BFT-M and BFT-AM, on 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**. Fig. 9 depicts the algorithm running time; colors indicate the number of seed sets, while line patterns indicate the algorithm. *Missing points (or curves) denote algorithms that timed out*. Note the logarithmic y axes.

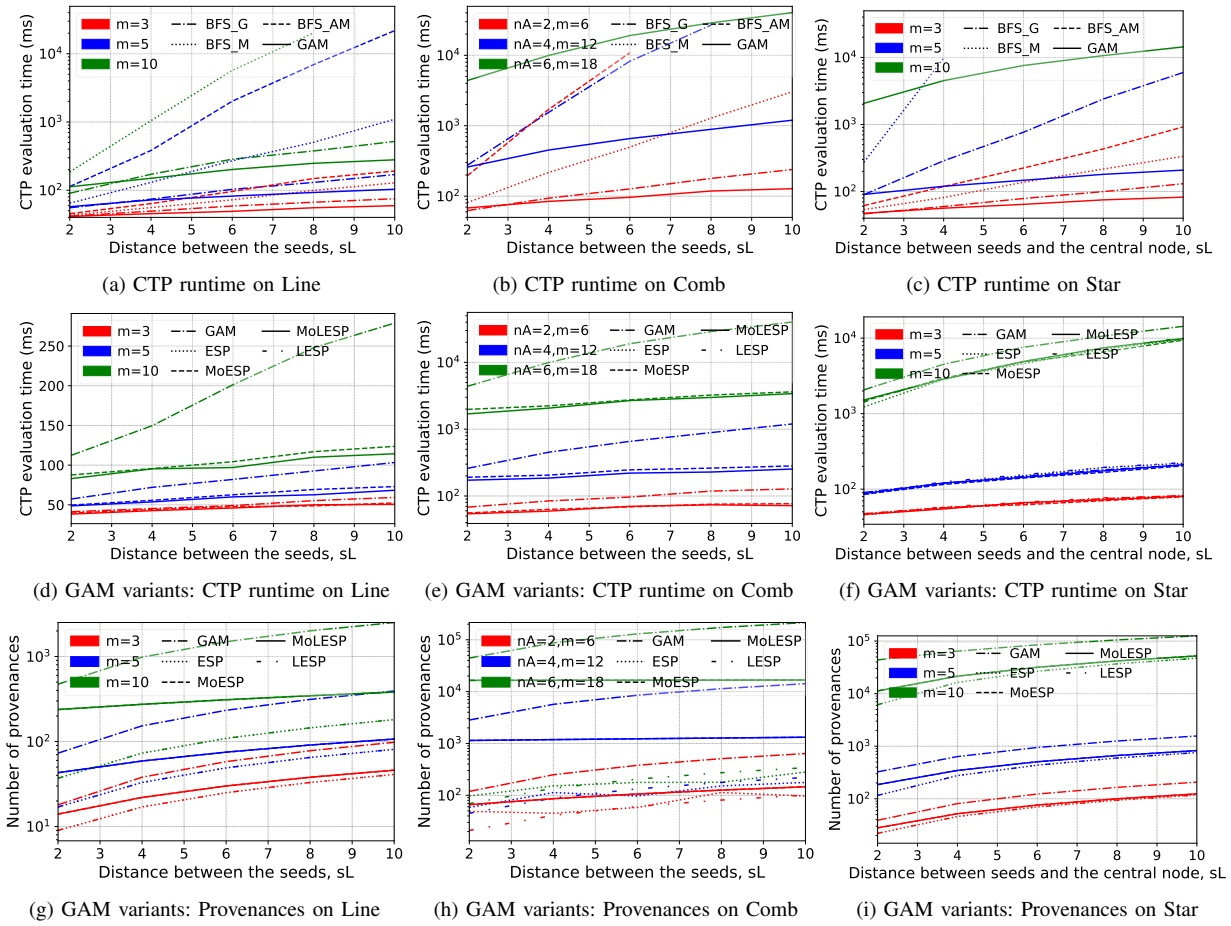
Across these plots, **BFT-M performs worse than BFT-AM**. On Line graphs, the difference is a factor $2\times$ for $m = 3$ and up to $100\times$ for $m = 10$. On the Comb and Star graphs, BFT-M times out on the larger graphs and queries. **BFT-AM takes even more than BFT-M**, by a factor of $15\times$, thus more executions timed out. **GAM is much faster** and completes execution in all cases. The reason, as explained in Sec. IV-A, 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*.

2) GAM algorithm variants: We now compare GAM with our proposed ESP, MoESP, LESP and MoLESP.

Fig. 9 shows the algorithm running times on Line, Comb and Star graphs; in all but Fig. 9d, the y axis is logarithmic. On Line and Comb graphs, **ESP and LESP failed to find results** due to edge set pruning, as explained in Sec. IV-D, thus the corresponding curves are missing.

The plots show, first, that edge set pruning significantly reduces the running time: **MoLESP is faster than GAM** by $1.3\times$ (Line graphs) up to $15\times$ (Comb graphs, nA=6, m=18). Second, on the Star graphs, where LESP (Sec. IV-F) 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. The number of provenances built by each algorithm is shown in Fig. 9g-9i; it is impacted by the pruning each algorithm applies. MoESP and MoLESP build the same number of provenances on Line and Comb graphs. The algorithm running times closely track the numbers of built provenances, highlighting the interest of controlling the latter through pruning.

Next, we compare these algorithms on our **Barabasi-Albert** graphs with a timeout of 25 minutes. For m=4, GAM timed out on even the smallest graph; all our proposed algorithms ran till completion. We report the algorithm running time until the last result is found (T_l) in Fig. 10a; Fig. 10b shows the

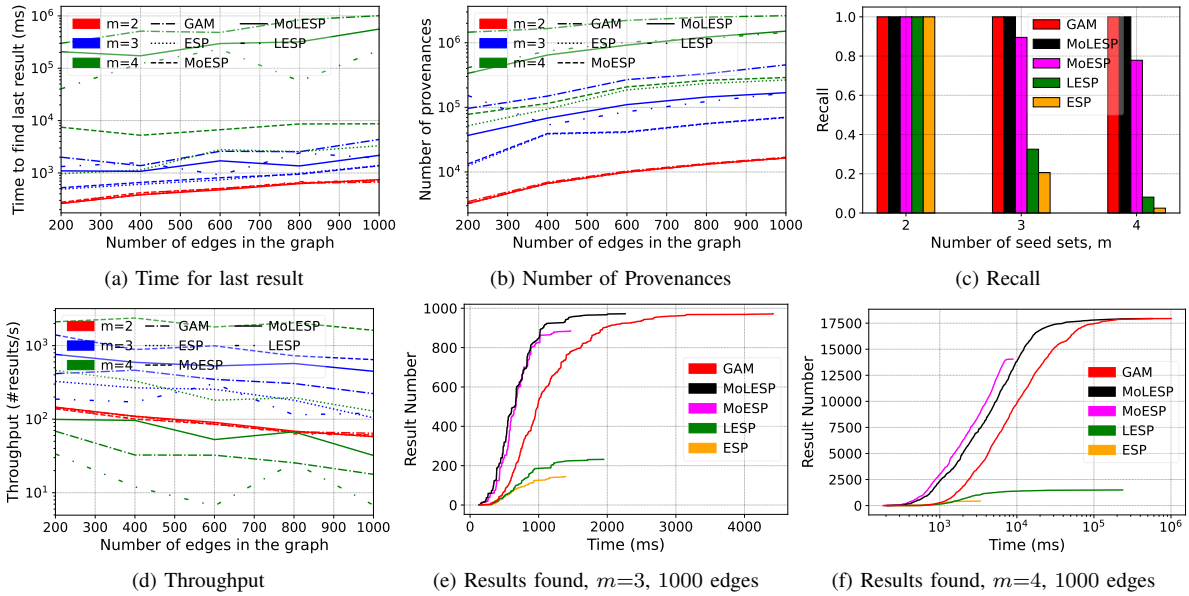


(a) CTP runtime on Line (b) CTP runtime on Comb (c) CTP runtime on Star

(d) GAM variants: CTP runtime on Line (e) GAM variants: CTP runtime on Comb (f) GAM variants: CTP runtime on Star

(g) GAM variants: Provenances on Line (h) GAM variants: Provenances on Comb (i) GAM variants: Provenances on Star

Fig. 9: Comparison of complete CTP evaluation baselines (top) and GAM variants (below) on synthetic benchmarks.



(a) Time for last result (b) Number of Provenances (c) Recall

(d) Throughput (e) Results found, m=3, 1000 edges (f) Results found, m=4, 1000 edges

Fig. 10: Comparison of GAM variants on Barabasi-Albert benchmarks.

number of provenances built by each algorithm (logarithmic y axis). Curves are non-monotonous due to variations in the randomly generated graphs. Fig. 10c plots the algorithm recall (fraction of results found), and Fig. 10d their throughput (results/running time). For m=2, all the variants have a full recall (ESP itself is complete, by Prop. 3). For m>2, the incomplete algorithms run faster than MoLESP but do not have a perfect recall, thus they also build less provenances.

MoLESP is at least 2× faster than GAM, with a perfect recall (even for m=4, where MoLESP may be incomplete), and it builds about 5× less provenances; this shows the interest of MoLESP pruning. MoESP has the highest throughput and finds more than 80% of the results; LESP has the lowest throughput with a recall of less than 0.2. For the largest graph, we also plot the number of results found, against the running time, for m=3 (Fig. 10e) and m=4 (Fig. 10f). MoLESP finds the results in about half the time as GAM; for the initial results, it closely follows MoESP. LESP takes longer in order to find its (incomplete) results. MoLESP has the advantage of MoESP and thus, a higher throughput and faster result finding than GAM; it is also pulled-down by LESP which is essential in order to guarantee completeness. Overall, **MoLESP is the best complete algorithm; among the incomplete ones, MoESP has the best throughput and recall.**

3) Comparison with QGSTP on real-world data: We now compare our best algorithm, MoLESP, with GAM and QGSTP [13] 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 restrictor (unidirectional exploration only), and ANY 1 selector to find just one result. Each QGSTP returned result is such that Prop. 9 ensures MoLESP finds it. Fig. 11a 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**.

E. Extended query (EQ) evaluation

1) Synthetic queries on CDF benchmark: We now compare our EQ evaluation system with the closest graph query evaluation baselines, on our CDF graphs (Sec. V-C) 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 Sec. III, 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, Fig. 11b 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 curves go farther at right). All missing points correspond to time-out.* JEDI succeeded only on the 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 completes in less than 2 minutes on the largest graph with 2.4M edges.

Query	JEDI	MoLESP	Virtuoso	Neo4j
J_1: 3 GPs, 2 CTPs	3.9	1.9	0.2	TIMEOUT
J_2: 2 GPs, 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 III: Query evaluation times (seconds) on YAGO3.

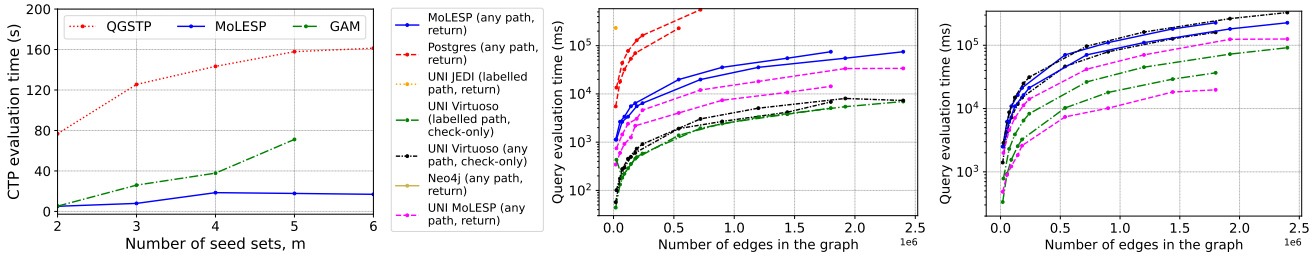
Fig. 11c shows similar results for m=3. Postgres timed-out in all cases. Virtuoso-SPARQL is 7× faster than Virtuoso-SQL; both return *non-minimal, duplicate results*. UNI-MoLESP outperforms every system, while *also returning connecting trees*. Note that the *bidirectional* MoLESP found about 7× more 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 GPs and the CTP (Sec. II). Despite the much larger search space due to bidirectionality, MoLESP scales well with the size of the graph and the cost of adding CTP to graph query engines is minimal.

2) Comparison with JEDI on real-world data: JEDI [18] used a set of (unidirectional, label-constrained) SPARQL 1.1 queries over YAGO3. Table III shows the queries' characteristics. We compare MoLESP similarly restricted (UNI and LABEL), on these queries, 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 Sec. IV-I, 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 GP evaluation and final joins.

This shows that **the optimizations described in Sec. IV-I make MoLESP robust also to large seed sets.**

VI. Related work

We focused on integrating *connecting tree patterns* (CTPs) into GPML [1], for which no parser or implementation is yet available [27]. Searching for connecting trees is not currently supported in any graph language. As outlined in Sec. III, languages such as SPARQL allow checking for paths connecting given nodes, but not returning them; others, including G-CORE [19], GPML, and Neo4j's Cypher return paths, however, the latter does not scale (Sec. V-E1). RPQProv [28] uses recursive SQL to return path labels; JEDI [18], [29] returns unidirectional paths (only). Many works focus on finding label-constrained paths between nodes [30]–[44], typically by using precomputed indexes or sketches. In our CTP evaluation algorithm, an index could be integrated by "reading from it" subtrees on which to Grow and Merge. Going beyond paths, *CTPs find trees connecting an arbitrary number of seed sets ($m\geq$3), traversing edges in any direction, independent of a scoring function*; we guarantee completeness for $m\leq$3 and finding a large set of results for arbitrary m. Path stitching leads to different results, which may require deduplication and minimization (Sec. III).



(a) GAM and MoLESP vs. QGSTP [13] on DBPedia.

(b) CDF benchmark performance for m=2, S_L∈{3,6}.

(c) CDF benchmark performance for m=3, S_L∈{3,6}.

Fig. 11: Comparison with state-of-the-art systems.

CTP evaluation is directly related to **keyword search in (semi-)structured data**; good surveys are [11], [12], [45]. The prior studies differ from ours: (*i*) [10], [46]–[54] are schema-dependent; (*ii*) [53], [55]–[57] assume available a compact summary of the graph; (*iii*) [7], [9], [23], [58] depend heavily on their score functions for pruning the search, particularly to approximate the best result [9], [58], [59] or return only top-k results [5], [8], [23], [48], [60]; (*iv*) [5], [6], [10], [23], [46], [61] are only unidirectional. For these reasons, they fail to meet our requirements (**R2**) to (**R5**) as outlined in Sec. I. MoLESP brings new, orthogonal, optimizations, and novel guarantees to the multi-threaded, C++ version [4] of GAM.

VII. Conclusions

We introduced CTPs, a new primitive for unstructured search in graphs, and showed how to integrate it within a graph query language such as GPML. We proposed novel algorithms which enumerate CTP results fully independently of a score function used to rank the CTP result trees. Some of our algorithms are guaranteed to find all results, but risk a high computational cost; others guarantee all results for at most 3 seed sets, or all results of certain shapes. MoLESP has the best trade-off between completeness and efficiency; our evaluation against the state-of-the-art systems demonstrates its performance. As part of our future work, we optimise the evaluation of multiple (batched) CTPs by sharing computations.

An intrinsic limitation of CTP (and EQ) evaluation is the potential search space explosion, when allowing edges in any direction, and/or when querying large, highly connected graphs. In such cases, depending on the user needs, a search method guaranteeing *some* answers, qualified wrt a specific cost metric, e.g., [13], [14], could be preferred.

References

[1] A. Deutsch, N. Francis, A. Green, K. Hare, B. Li, L. Libkin, T. Lindaaker, V. Marsault, W. Martens, J. Michels, F. Murlak, S. Plantikow, P. Selmer, O. van Rest, H. Voigt, D. Vrgoc, M. Wu, and F. Zemke, "Graph pattern matching in GQL and SQL/PGQ," in *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, 2022, pp. 2246–2258. [Online]. Available: https://doi.org/10.1145/3514221.3526057

[2] S. Sakr, A. Bonifati, H. Voigt, A. Iosup, K. Ammar, R. Angles, W. G. Aref, M. Arenas, M. Besta, P. A. Boncz, K. Daudjee, E. D. Valle, S. Dumbrava, O. Hartig, B. Haslhofer, T. Hegeman, J. Hidders, K. Hose, A. Iamnitchi, V. Kalavri, H. Kapp, W. Martens, M. T. Özsu, E. Peukert, S. Plantikow, M. Ragab, M. Ripeanu, S. Salihoglu, C. Schulz, P. Selmer, J. F. Sequeda, J. Shinavier, G. Szárnyas, R. Tommasini, A. Tumeo, A. Uta, A. L. Varbanescu, H. Wu, N. Yakovets, D. Yan, and E. Yoneki, "The future is big graphs: a community view on graph processing systems," *Commun. ACM*, vol. 64, no. 9, pp. 62–71, 2021. [Online]. Available: https://doi.org/10.1145/3434642

[3] W3C, "SPARQL 1.1," 2013. [Online]. Available: https://www.w3.org/TR/sparql11-overview/

[4] A.-C. Anadiotis, O. Balalau, T. Bouganim, F. Chimienti, H. Galhardas, M. Y. Haddad, S. Horel, I. Manolescu, and Y. Youssef, "Empowering Investigative Journalism with Graph-based Heterogeneous Data Management," *Bulletin of the Technical Committee on Data Engineering*, Sep. 2021. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03337650

[5] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "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*, 2002, pp. 431–440. [Online]. Available: https://doi.org/10.1109/ICDE.2002.994756

[6] B. Aditya, G. Bhalotia, S. Chakrabarti, A. Hulgeri, C. Nakhe, Parag, and S. Sudarshan, "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*, 2002, pp. 1083–1086. [Online]. Available: http://www.vldb.org/conf/2002/S33P11.pdf

[7] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, "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*, 2005, pp. 505–516. [Online]. Available: http://www.vldb.org/archives/website/2005/program/paper/wed/p505-kacholia.pdf

[8] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou, "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*, 2008, pp. 903–914. [Online]. Available: https://doi.org/10.1145/1376616.1376706

[9] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin, "Finding top-k min-cost connected trees in databases," in *ICDE*, R. Chirkova, A. Dogac, M. T. Özsu, and T. K. Sellis, Eds. IEEE Computer Society, 2007, pp. 836–845. [Online]. Available: https://doi.org/10.1109/ICDE.2007.367929

[10] S. Agrawal, S. Chaudhuri, and G. Das, "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*, R. Agrawal and K. R. Dittrich, Eds. IEEE Computer Society, 2002, pp. 5–16. [Online]. Available: https://doi.org/10.1109/ICDE.2002.994693

[11] H. Wang and C. C. Aggarwal, "A survey of algorithms for keyword search on graph data," in *Managing and Mining Graph Data*, ser. Advances in Database Systems, C. C. Aggarwal and H. Wang, Eds. Springer, 2010, vol. 40, pp. 249–273. [Online]. Available: https://doi.org/10.1007/978-1-4419-6045-0_8

[12] J. Coffman and A. C. Weaver, "An empirical performance evaluation of relational keyword search techniques," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 1, pp. 30–42, 2014. [Online]. Available: https://doi.org/10.1109/TKDE.2012.228

[13] Y. Shi, G. Cheng, T. Tran, E. Kharlamov, and Y. Shen, "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*, J. Leskovec, M. Grobelnik, M. Najork, J. Tang, and L. Zia, Eds. ACM / IW3C2, 2021, pp. 1410–1421, code available at: https://github.com/nju-websoft/QGSTP. [Online]. Available: https://doi.org/10.1145/3442381.3449900

[14] Y. Sun, X. Xiao, B. Cui, S. K. Halgamuge, T. Lappas, and J. Luo, "Finding group steiner trees in graphs with both vertex and edge weights," *Proc. VLDB Endow.*, vol. 14, no. 7, pp. 1137–1149, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p1137-sun.pdf

[15] A. G. Anadiotis, O. Balalau, C. Conceição, H. Galhardas, M. Y. Haddad, I. Manolescu, T. Merabti, and J. You, "Graph integration of structured, semistructured and unstructured data for data journalism," *Inf. Syst.*, vol. 104, p. 101846, 2022. [Online]. Available: https://doi.org/10.1016/j.is.2021.101846

[16] I. F. Cruz, A. O. Mendelzon, and P. T. Wood, "A graphical query language supporting recursion," in *SIGMOD*, U. Dayal and I. L. Traiger, Eds. ACM Press, 1987, pp. 323–330. [Online]. Available: https://doi.org/10.1145/38713.38749

[17] P. Barceló, L. Libkin, and J. L. Reutter, "Querying regular graph patterns," *J. ACM*, vol. 61, no. 1, pp. 8:1–8:54, 2014. [Online]. Available: https://doi.org/10.1145/2559905

[18] C. Aebeloe, G. Montoya, V. Setty, and K. Hose, "Discovering diversified paths in knowledge bases," *Proc. VLDB Endow.*, vol. 11, no. 12, pp. 2002–2005, 2018, code available at: http://qweb.cs.aau.dk/jedi/. [Online]. Available: http://www.vldb.org/pvldb/vol11/p2002-aebeloe.pdf

[19] R. Angles, M. Arenas, P. Barceló, P. A. Boncz, G. H. L. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. F. Sequeda, O. van Rest, and H. Voigt, "G-CORE: A core for future graph query languages," in *SIGMOD*, G. Das, C. M. Jermaine, and P. A. Bernstein, Eds. ACM, 2018, pp. 1421–1432. [Online]. Available: https://doi.org/10.1145/3183713.3190654

[20] C. Ordonez, "Optimization of linear recursive queries in SQL," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 2, pp. 264–277, 2010. [Online]. Available: https://doi.org/10.1109/TKDE.2009.83

[21] N. Tziavelis, W. Gatterbauer, and M. Riedewald, "Beyond equi-joins: Ranking, enumeration and factorization," *Proc. VLDB Endow.*, vol. 14, no. 11, pp. 2599–2612, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p2599-tziavelis.pdf

[22] Neo4j, "Cypher Query Language," 2022. [Online]. Available: https://neo4j.com/developer/cypher/

[23] H. He, H. Wang, J. Yang, and P. S. Yu, "BLINKS: ranked keyword searches on graphs," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, 2007, pp. 305–316. [Online]. Available: https://doi.org/10.1145/1247480.1247516

[24] V. M. S. and J. R. Haritsa, "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*, 2019, pp. 103–111. [Online]. Available: https://doi.org/10.1145/3297001.3297014

[25] ——, "Operator implementation of result set dependent KWS scoring functions," *Inf. Syst.*, vol. 89, p. 101465, 2020. [Online]. Available: https://doi.org/10.1016/j.is.2019.101465

[26] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, 1999.

[27] A. Deutsch, N. Francis, L. Libkin, and V. Marsault, "Email communication with GPML authors," 2022, initiated: 2022-08-08.

[28] S. C. Dey, V. Cuevas-Vicenttín, S. Köhler, E. Gribkoff, M. Wang, and B. Ludäscher, "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*, G. Guerrini, Ed. ACM, 2013, pp. 214–223. [Online]. Available: https://doi.org/10.1145/2457317.2457353

[29] C. Aebeloe, V. Setty, G. Montoya, and K. Hose, "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*, ser. CEUR Workshop Proceedings, M. van Erp, M. Atre, V. López, K. Srinivas, and C. Fortuna, Eds., vol. 2180. CEUR-WS.org, 2018. [Online]. Available: http://ceur-ws.org/Vol-2180/paper-01.pdf

[30] K. Anyanwu, A. Maduko, and A. P. Sheth, "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*, 2007, pp. 797–806. [Online]. Available: https://doi.org/10.1145/1242572.1242680

[31] A. Gubichev and T. Neumann, "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*, 2011. [Online]. Available: http://webdb2011.rutgers.edu/papers/Paper21/pathwebdb.pdf

[32] A. Gubichev, S. J. Bedathur, and S. Seufert, "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*, 2013, p. 14. [Online]. Available: http://event.cwi.nl/grades2013/14-gubichev.pdf

[33] G. H. L. Fletcher, J. Peters, and A. Poulovassilis, "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*, E. Pitoura, S. Maabout, G. Koutrika, A. Marian, L. Tanca, I. Manolescu, and K. Stefanidis, Eds. OpenProceedings.org, 2016, pp. 636–639. [Online]. Available: https://doi.org/10.5441/002/edbt.2016.67

[34] N. Yakovets, P. Godfrey, and J. Gryz, "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*, 2016, pp. 1875–1889. [Online]. Available: https://doi.org/10.1145/2882903.2882944

[35] L. D. J. Valstar, G. H. L. Fletcher, and Y. Yoshida, "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*, 2017, pp. 345–358. [Online]. Available: https://doi.org/10.1145/3035918.3035955

[36] S. Wadhwa, A. Prasad, S. Ranu, A. Bagchi, and S. Bedathur, "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*, 2019, pp. 1463–1480. [Online]. Available: https://doi.org/10.1145/3299869.3319882

[37] J. Kuijpers, G. Fletcher, T. Lindaaker, and N. Yakovets, "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*, 2021, pp. 582–587. [Online]. Available: https://doi.org/10.5441/002/edbt.2021.68

[38] Y. Peng, X. Lin, Y. Zhang, W. Zhang, and L. Qin, "Answering reachability and k-reach queries on large graphs with label constraints," *VLDB J.*, vol. 31, no. 1, pp. 101–127, 2022. [Online]. Available: https://doi.org/10.1007/s00778-021-00695-0

[39] D. Arroyuelo, A. Hogan, G. Navarro, and J. Rojas-Ledesma, "Time- and space-efficient regular path queries on graphs," in *38th IEEE International Conference on Data Engineering, ICDE 2022*. IEEE, 2022.

[40] I. Na, I. Yi, K. Whang, Y. Moon, and S. J. Hyun, "Regular path query evaluation sharing a reduced transitive closure based on graph reduction," in *38th IEEE International Conference on Data Engineering, ICDE 2022*. IEEE, 2022.

[41] K. Hao, L. Yuan, and W. Zhang, "Distributed hop-constrained s-t simple path enumeration at billion scale," *Proc. VLDB Endow.*, vol. 15, no. 2, pp. 169–182, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol15/p169-hao.pdf

[42] X. Chen, Y. Peng, S. Wang, and J. X. Yu, "DLCR: efficient indexing for label-constrained reachability queries on large dynamic graphs," *Proc. VLDB Endow.*, vol. 15, no. 8, pp. 1645–1657, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p1645-chen.pdf

[43] Q. Chen, O. Lachish, S. Helmer, and M. H. Böhlen, "Dynamic spanning trees for connectivity queries on fully-dynamic undirected graphs," *Proc. VLDB Endow.*, vol. 15, no. 8, pp. 1645–1657, 2022.

[44] C. Zhang, A. Bonifati, H. Kapp, V. I. Haprian, and J. Lozi, "A reachability index for recursive label-concatenated graph queries," *ICDE*, 2023.

[45] J. Yang, W. Yao, and W. Zhang, "Keyword search on large graphs: A survey," *Data Sci. Eng.*, vol. 6, no. 2, pp. 142–162, 2021. [Online]. Available: https://doi.org/10.1007/s41019-021-00154-4

[46] V. Hristidis and Y. Papakonstantinou, "DISCOVER: keyword search in relational databases," in *VLDB*, 2002. [Online]. Available: http://www.vldb.org/conf/2002/S19P02.pdf

[47] V. Hristidis, L. Gravano, and Y. Papakonstantinou, "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*, 2003, pp. 850–861. [Online]. Available: http://www.vldb.org/conf/2003/papers/S25P03.pdf

[48] Y. Luo, X. Lin, W. Wang, and X. Zhou, "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*, 2007, pp. 115–126. [Online]. Available: https://doi.org/10.1145/1247480.1247495

[49] Y. Luo, W. Wang, X. Lin, X. Zhou, J. Wang, and K. Li, "SPARK2: top-k keyword query in relational databases," *IEEE Trans. Knowl. Data Eng.*, vol. 23, no. 12, pp. 1763–1780, 2011. [Online]. Available: https://doi.org/10.1109/TKDE.2011.60

[50] P. de Oliveira, A. S. da Silva, E. S. de Moura, and R. Rodrigues, "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*, 2018, pp. 1344–1347. [Online]. Available: https://doi.org/10.1109/ICDE.2018.00146

[51] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "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*, 2003, pp. 16–27. [Online]. Available: https://doi.org/10.1145/872757.872762

[52] V. Hristidis, Y. Papakonstantinou, and A. Balmin, "Keyword proximity search on XML graphs," in *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, 2003, pp. 367–378. [Online]. Available: https://doi.org/10.1109/ICDE.2003.1260806

[53] T. Tran, H. Wang, S. Rudolph, and P. Cimiano, "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*, 2009, pp. 405–416. [Online]. Available: https://doi.org/10.1109/ICDE.2009.119

[54] G. Kasneci, M. Ramanath, M. Sozio, F. M. Suchanek, and G. Weikum, "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*, 2009, pp. 868–879. [Online]. Available: https://doi.org/10.1109/ICDE.2009.64

[55] A. Balmin, V. Hristidis, and Y. Papakonstantinou, "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*, 2004, pp. 564–575. [Online]. Available: http://www.vldb.org/conf/2004/RS15P2.PDF

[56] W. Le, F. Li, A. Kementsietsidis, and S. Duan, "Scalable keyword search on large RDF data," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 11, pp. 2774–2788, 2014. [Online]. Available: https://doi.org/10.1109/TKDE.2014.2302294

[57] Y. Zhu, Q. Zhang, L. Qin, L. Chang, and J. X. Yu, "Cohesive subgraph search using keywords in large networks," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 1, pp. 178–191, 2022. [Online]. Available: https://doi.org/10.1109/TKDE.2020.2975793

[58] R. Li, L. Qin, J. X. Yu, and R. Mao, "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*, 2016, pp. 91–106. [Online]. Available: https://doi.org/10.1145/2882903.2915217

[59] M. Kargar, L. Golab, D. Srivastava, J. Szlichta, and M. Zihayat, "Effective keyword search over weighted graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 2, pp. 601–616, 2022. [Online]. Available: https://doi.org/10.1109/TKDE.2020.2985376

[60] Y. Yang, D. Agrawal, H. V. Jagadish, A. K. H. Tung, and S. Wu, "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*, 2019, pp. 338–349. [Online]. Available: https://doi.org/10.1109/ICDE.2019.00038

[61] Z. Zhang, J. X. Yu, G. Wang, Y. Yuan, and L. Chen, "Key-core: cohesive keyword subgraph exploration in large graphs," *World Wide Web*, vol. 25, no. 2, pp. 831–856, 2022. [Online]. Available: https://doi.org/10.1007/s11280-021-00926-y