There are Trillions of Little Forks in the Road. Choose Wisely!

– Estimating the Cost and Likelihood of Success of Constrained Walks to Optimize a Graph Pruning Pipeline –

Nicolas Tripoul\textsuperscript{1}, Hassan Halawa\textsuperscript{1}, Tahsin Reza\textsuperscript{1}, Geoffrey Sanders\textsuperscript{2}, Roger Pearce\textsuperscript{2}, Matei Ripeanu\textsuperscript{1}
\textsuperscript{1}Department of Electrical and Computer Engineering, University of British Columbia
\textsuperscript{2}Center for Applied Scientific Computing, Lawrence Livermore National Laboratory
\{mtripoul, hhalawa, treza\}@ece.ubc.ca \{sanders29, rpearce\}@llnl.gov matei@ece.ubc.ca

Abstract—We have developed \cite{1} a highly scalable algorithmic pipeline for pattern matching in labeled graphs and demonstrated it on trillion-edge graphs. This pipeline: (i) supports arbitrary search patterns, (ii) identifies all the vertices and edges that participate in matches – offering 100\% precision and recall, and (iii) supports realistic data analytics scenarios. This pipeline is based on graph pruning; it decomposes the search template into individual constraints and uses them to repeatedly prune the graph to a final solution.

Our current solution, however, makes a number of ad-hoc intuition-based decisions with impact on performance. In a nutshell these relate to (i) constraint selection – which constraints to generate? (ii) constraint ordering – in which order to use them? and (iii) individual constraint generation – how to best verify them? This position paper makes the observation that just two primitives: estimating the runtime cost and likelihood of success of a constrained walk in a labeled graph one can inform these optimization decisions. We propose a preliminary solution to make these estimates, and demonstrate – using a prototype shared-memory implementation – that this: (i) is feasible with low overheads, and (ii) offers accurate enough information to optimize our pruning pipeline by a significant margin.

I. INTRODUCTION

Pattern matching in graphs, that is, finding subgraphs that match a small template graph within a large background graph (Fig. 1) is fundamental to graph analysis and has applications in multiple areas such as social network analysis \cite{2}, bioinformatics \cite{3}, and information mining \cite{4}. A match can be broadly categorized as either exact - i.e., there is a bijective map between the vertices/edges in the template and those in the matching subgraph, or approximate - the template and the match are just similar by some defined similarity metric \cite{5}. Berry et al. \cite{6} introduced the problem of type-isomorphism: metadata graphs where vertices and edges are labeled and, in addition to topological constraints, a match identifies nodes and edges with the same labels in the template and the background graph. While the labeled version does not reduce the complexity of the problem (thought to be NP), past experience \cite{2}, \cite{7}, \cite{8} has demonstrated that exact pattern matching in labeled graphs can be a powerful tool with potential for practical, real-world applications.

We have demonstrated \cite{1}, \cite{9} that an algorithmic pipeline for pattern matching based on graph pruning is scalable and effective. Our pruning-based approach is motivated by viewing the search template as specifying a set of constraints the vertices and edges that participate in a match must meet. Our pipeline iterates over these constraints to eliminate all the vertices and edges that do not participate in any match. The intuition for the effectiveness of this technique is the following: traditional tree-search techniques \cite{10} generally attempt to enumerate all matches through explicit search (often depth-first search). When a search path fails, such an unprofitable path is marked invalid and ignored in the subsequent search steps. Our experience shows that often it is much cheaper to focus on iteratively eliminating the vertices and edges that do not meet the label and topological constraints introduced by the search template. We have developed techniques to generate increasingly complex constraints from the search template and to use them to prune the graph to a precise solution that is the union of all matches.

Contributions. Our current pipeline (detailed in §II), however, uses a number of ad-hoc intuition-based heuristics for constraint selection, ordering, and generation (outlined in §V-A). As these heuristics have a sizable impact on performance, yet they are only intuition-based, we explore whether a better solution is possible. To this end, this position paper makes the observation that just two primitives: estimating constraint selectivity and cost (i.e., the likelihood of success and the runtime cost of deciding whether a constrained walk in the background graph that matches the constraint exists) are sufficient to design informed heuristics. Multiple optimization heuristics can use these two estimates (§III). We outline the challenges for making those estimates (§IV), and propose a first solution to compute them based on a stochastic graph model (§V-B). Our preliminary results (§VI) suggest that, although based on an arguably simplistic graph model, our technique enables computing the needed estimates: (i) with low runtime overheads, and (ii) with enough accuracy to optimize our pruning pipeline by a significant margin.
We postulate that our approach for estimating constraint cost and selectivity is useful beyond the main application that we explore in this paper (i.e., accelerating exact pattern matching). As a first direct extension, the same approach and estimators can be adapted to accelerate approximate pattern matching.

II. BACKGROUND: THE PRUNING PIPELINE
Our pipeline extracts an increasingly complex set of constraints from the search pattern and uses them to prune the graph to a precise solution. The result is the union of all matches (or, otherwise said, the subgraph that contains all – and only – the vertices and edges that are part of a match). The pipeline can also be used to enumerate all matches.

As the algorithmic details (presented in detail in [1]) are complex, we present here only high-level information to understand the pruning pipeline and the optimization problems that arise. To this end, Fig. 2 illustrates the complete workflow for the background graph and search template in Fig. 1 for which constraint generation is detailed in Fig. 3.

In our vertex-centric formulation for pruning, a vertex must satisfy two types of constraints: local and non-local, to possibly be part of a match. Local constraints involve only the vertex and its neighborhood: a vertex in an exact match needs to (i) match the label of a corresponding vertex in the template, and (ii) have edges to vertices labeled as prescribed in the adjacency structure of this corresponding vertex in the template. Local constraints are checked by having vertices communicate their (provisional) template match with their one-hop neighbors in the current solution subgraph (i.e., the currently pruned graph). We call this process Local Constraint Checking (LCC). Our experiments [1] show that, while LCC is responsible for removing the bulk of non-matching vertices and edges, for non-trivial templates only a small part of the runtime is spent here.

Non-trivial templates require additional checks for non-local properties to guarantee that all non-matching vertices are eliminated. (Fig. 4 highlights the need for these additional checks). Non-Local Constraint Checking (NLCC) works as follows: first, based on the search template $G_0$, we generate the set of constraints that are to be verified, then prune the graph using them. Key choices that impact performance are: (i) constraint selection – which constraints to generate?; (ii) constraint ordering – in which order to use them?; and (iii) individual constraint generation – how to best verify them?

Checking non-local constraints is expensive as it involves determining whether a constrained walk, a walk that matches the constraint and starts at a specified vertex, exists in the background graph. To this end, the same algorithmic approach is used regardless of constraint type: the constrained walk is initiated from the background graph vertices whose corresponding template vertices are identified to need to match the constraint. Depending on the implementation the search for the constrained walk progresses depth- or breadth-first in the background graph. If a constrained walk is found to exist then the start vertex is valid, otherwise it is pruned away. We have implemented this search for constrained walks using a token passing mechanism for a distributed platform, and as a parallel depth-first search on a shared-memory platform. Along the search path, the system verifies that all expected labels are encountered and, where necessary, uses the accumulated information to verify that different/repeated vertex identity constraint expectations are met.

Constraint types. We have identified three types of non-local constraints: (i) Cycle Constraints (CC), (ii) Path Constraints (PC), and (iii) constraints that require Template-Driven Search (TDS). TDS constraints are based on aggregating multiple paths/cycles and enable further pruning. When based on the full template (what we call a TDS* constraint) they can be used for match enumeration and to ensure, in the general case, that pruning yields no false positives.

Starting Vertices. For CC and TDS constraints, in the current solution, the constrained walk must be initiated from each vertex in the background graph that may participate in the substructure, whereas for PC only from terminal vertices.

Heuristics. Section V-A presents the current heuristics to select, order, and generate constraints. Fig. 3 shows the progress of non-local constraint generation and graph pruning.

Termination. LCC and NLCC are applied in sequence to iteratively prune the graph. Pruning terminates when all generated non-local constraints have been verified and no vertices are eliminated during the subsequent LCC phase.

III. OPPORTUNITIES FOR OPTIMIZATION
We summarize the areas where the current pruning framework used ad-hoc heuristics – developed based on our intuition – and where informed heuristics are likely needed:

- Constraint selection. The set of constraints that can be generated is large – even from a medium complexity pattern. Selecting a ‘good’ set of constraints, rather than using all of them, has a key impact on performance. For example, our experiments [1] show that selecting a set of simpler constraints (i.e., CC and PC) first instead of exclusively selecting TDS constraints is extremely effective: for some patterns (e.g., RDT-1 in Fig. 5) the system is not able to complete in a reasonable time without these simpler constraints being selected first, for other patterns this selection enables a 2.5× speedup. The current heuristic used for constraint selection is presented in Section V-A.

- Constraint ordering. Once the set of constraints is selected, one needs to decide the order in which these are used to prune the graph. Our current heuristic is to check path-, then cycle-, then TDS- constraints, and within each class to order constraints by size. This heuristic is still underdetermined (as multiple constraints may have the same length), and, as we show in §VI, sub-optimal.

1Trivial templates are processed relatively fast and there is no need for better heuristics. Trivial templates are those that (i) are acyclic and do not have duplicate labels – can be supported only using LCC; or (ii) edge-monocyclic (i.e., each edge belongs to at most one cycle) and do not have duplicate labels – can be supported with LCC and cycle-checking only.

2For example, for the relatively complex IMDB-1 search template in Fig. 5 over 98% of the runtime is spent checking these constraints [9].
Fig. 2: Algorithm walk-through for the example background graph and template in Fig. 1, depicting (in grey) which vertices and edges in the current solution graph \( G^* = \langle V^*, E^* \rangle \) are eliminated during each iteration. The algorithm starts by checking the local constraints (LCC), then it iterates over all non-local constraints (NLCC) and, after each of them checks the local constraints again. The NLCC constraints extracted from template \( G_3 \) are listed in Fig. 3. For clarity, the example does not show the application of some of the constraints in Fig. 3 that do not eliminate vertices or edges. Figure reused from [1].

Fig. 3: A high-level depiction of non-local constraint generation for the template in Fig. 1. The figure shows the steps to generate the required cycle checking (CC), path checking (PC) and template-drive search (TDS) constraints. Due to limited space, the figure presents only a subset of the path and TDS constraints generated. Figure reused from [1].

Fig. 4: Three examples of search templates and background graphs that justify the full set of pruning constraints. Template (a) is a 3-cycle; cycles of length \( k \) with repeated labels in the background graph which meet neighborhood constraints, surviving LCC. Template (b) contains several vertices with non-unique labels; to its right there is a background graph that meets individual point-to-point path constraints, also surviving NLCC-path checking. Template (c) is characterized by two 4-cliques that overlap at a 3-cycle; the background graph structure to the right is doubly periodic (a \( 4 \times 3 \) torus) and meets all edge and vertex cycle constraints, surviving NLCC-cycle checking. Templates (b) and (c) require template-driven search to guarantee no false positives; template (a) only needs cycle-checking in addition to LCC. Figure reused from [1].

- **Optimize individual constraint verification.** To check a constraint, one verifies whether a constrained walk based on the constraint exists in the background graph. There are multiple ways in which these constrained walks can be searched for: for example, for a path constraint, one can start the walk at either end of the constraint; for a cycle constraint one can start at any point of the cycle and go in either direction. Our current solution does not attempt any optimization: just picks one option at random.

- **Decide when to switch from pruning to direct enumeration.** If an enumeration of all matches (and not the pruned graph, i.e., the union of all matches) is the end-goal then early switching to enumeration based on the full template (what we call a TDS* constraint) and stopping the pruning can bring high benefits. Our current solution is unable to make this decision and always prunes down to a precise solution before enumeration.

- **Decide whether pruning is useful at all.** For all (realistic – we believe) scenarios we have experimented with so far, pruning had been faster than a solution based on direct search. However, one can construct use-cases where direct search for the template, similar to enumeration, will perform better. The intuition is that pruning reduces the combinatorial explosion direct search encounters. In cases where such combinatorial explosion does not appear – e.g., searching for a pattern that includes labels that are unique in the background graph – direct search will perform better. The goal is to predict which approach performs better.

- **Understand when to trigger load-(re)balancing.** We have evidence that load-imbalances can be an issue, and that load-(re)balancing by randomly reshuffling the graph is effective. Our decision to trigger this is, however, backward looking: we check for imbalances after pruning a number of constraints (and do not attempt to predict load imbalances) based on the current state of the pruned graph and the next constraint(s) to use. We postulate that estimating the cost of running future constraints can lead to better load balancing decisions.

**Key intuition.** We observe that the key to making informed decisions on all the above issues is estimating: (i) the **selectivity** of a constraint - i.e., number of vertices eliminated by the constraint, and (ii) the **cost** to run the constraint - the time to determine whether a constrained walk that matches the constraint and starts at a specified vertex exists in the background graph.

**Estimating Selectivity.** that is, the number of vertices eliminated by the constraint.

- **Real graphs are hard to model.** Known graph models may be used as an approximation, however the approximation embedded in the model becomes a source of error for any derived estimates.

- **There is a trade-off between, on the one side, model accuracy – and its ability to model real world graphs with high fidelity – and, on the other side, mathematical tractability for the derived metrics we are interested in.**

- **There is an additional trade-off between, on the one side, model complexity – the volume of information needed –
and, on the other side, effort to gather and maintain this data as the graph gets pruned given the need to dynamically update data after running each constraint.

Estimating the Cost of running a constraint, that is, the cost of searching for a constrained walk that matches the constraint:

- Optimization-specific aspects. Optimizations beyond a naive depth-first search implementation can have a sizable impact on cost and yet are hard to model accurately. We describe the optimizations we implemented for our prototype shared memory pipeline in §V-D. Our evaluation shows how they have a key impact on performance: between 4× and 10× improvement (Fig. 6 in §VI-B). Given their magnitude, it is key to model these optimizations.

- Architecture-specific aspects. Accurately estimating costs is highly dependent on the architecture. On the one side, each design is extended with implementation-level optimizations that are highly architecture dependent. On the other side, the bottleneck resource is generally architecture dependent – e.g., memory access / latency for shared memory architectures vs. generated network traffic for a distributed architecture – thus, different models may be needed for different architectures.

V. SOLUTION DESIGN

This section presents our solution to optimize constraint selection, ordering, and verification. We focus on the performance improvement achievable through a real-time analysis of the background graph and search template topology.

A. Prior Approach

Our previous approach was intuitive and experimental [1]: checking shorter constraints should be prioritized, path constraints are faster than cycle constraints which are also faster than TDS constraints, and – where applicable – the constrained walk should have a vertex with low frequency label at or near the beginning of the walk. The rest of this section presents our past heuristics for non-local constraint generation in detail.

For templates with unique labels and mono-cyclic edges, LCC and CC are sufficient to generate a precise solution. For the general case, we generate non-local constraints using the following heuristic: first, all the leaf vertices with unique labels are identified and ignored from this process (as LCC guarantees pruning if there is no match). Next, if the template has cycles, the individual cycles are identified (e.g., Step 3 in Fig. 3) and a cycle constraint is generated for each minimal cycle. Next, vertices with identical labels are identified and all path constraints are generated for all such pairs except for pairs that belong to one of the cycles constraints generated above as they are already verified by CC (e.g., the pentagonal vertices identified in Step 4 in Fig. 3).

Finally, we identify TDS constraints in three steps: first, for non-edge-monocyclic templates, a TDS cyclic constraint is generated through the union of the previously identified cycle constraints sharing at least one edge. This results in a higher-order cyclic structure with a maximal set of edges that covers all the non-edge-mono cycles (e.g., Step 5(1) in Fig. 3). Second, for templates with repeated labels, a new TDS constraint is generated through a union of all previously identified path constraints sharing at least a vertex. This procedure generates higher order structure that covers all the template vertices with repeated labels (e.g., Step 5(2) in Fig. 3). The third step generates a TDS constraint as the union of the previous two constraints sharing at least a constraint (e.g., Step 5(3) in Fig. 3). It can be shown that the third step with the local constraint eliminates all false positives [1]. The final step is to generate the TDS constraint from the entire template pattern, this is used to enumerate matches 3.

B. Proposed Approach: Intuition

We aim to minimize pruning time (maximize pruning rate). Each individual constraint can be thought of as having its own pruning rate – or effectiveness. Constraint effectiveness can be thought of as the ratio between the number of vertices it will prune (its selectivity) to the time it will take to find a constrained walk that satisfies the constraint (its cost). Once we estimate constraint effectiveness, we develop heuristics for the optimization problems mentioned in §III.

The number of vertices pruned (i.e., the selectivity) for a specific constraint can be found from the probability that it will succeed (i.e., the constrained walk exists). This is tied to the size of the search tree that can be explored, the different labels that are part of the constraint relative to their frequency in the background graph, and the additional requirements that need to be met such as cycles. As a general rule, the shorter the constraint the more probable it is that a walk exists, however less frequent labels will decrease this probability.

The runtime cost of finding out whether a constraint is satisfied for a vertex is harder to estimate. It depends on the specific system architecture and the different implementation optimizations. As a first-order approximation, to estimate cost for our shared memory implementation, we attempt to estimate the number of memory accesses generated since graph algorithms have generally poor locality and for which memory access latency is the main driver of runtime. This approach ties the estimated cost to the number of edges and the number of vertices that are traversed during the search.

Implementation optimizations have to be taken into account to allow for an accurate estimation of the time it takes to run a constraint. Our shared memory implementation includes three key optimizations: (i) Early Termination ensures that if a constrained walk is found the, possibly parallel, depth-first search stops; (ii) Work Aggregation makes sure that vertices are not visited multiple times during the depth-first search; and (iii) Multiple Vertex Validation, as in the original implementation designed for a distributed platform, validates each vertex that may participate in a sub-structure independently. However if validation succeeds for one vertex, all the others are validated and the search terminates early. The impact of these optimizations is evaluated in §VI-B.

3: The constraint generation code is available at https://github.com/NTripoul/patternmatching-generator. The code generates the different NLCC constraints that are available.
C. Graph Model and Computing the Estimators

Our mindset is framed by an aphorism attributed to G. Box: “All models are wrong but some are useful”. We start from a mathematically tractable – yet arguably simplistic – graph model, analytically derive properties of interest, and evaluate (in the following section) whether these derivations are useful when applied to our scenarios. Our graph model is based on the following two hypotheses:

- Vertices’ labels determine their connectivity. For each label, the vertices with this label have a degree close to the average for that label.
- Edges between labels are uniformly distributed.

We acknowledge that in most real-world cases, this model is not an accurate representation of the background graph: in many cases the degree distribution is power-law, hence, the correlation between label and node degree that we model may not exist, and a correlation between vertex label and the labels of its neighbors (which we do not model) may exist. However, this graph model has the advantage that it is simple enough so that the estimates for the properties of interest, cost and selectivity, can be derived analytically. We feed the analytically derived estimates with data derived from fitting the graph at hand to the model.

Given the random graph model, the properties of interest can be derived. For example, we can estimate the probability that a vertex with a given label has a neighbor with another given label, based on the uniform distribution of edges hypothesis, and from this, the probability that a constrained walk exists can be derived. This allows us to estimate the cost of running a specific depth-first search, and the probability that this search will succeed. Due to lack of space, the derivation of the estimates is detailed in Appendix A.

One observation is key to the feasibility of this approach: the statistics that need to be gathered and maintained are proportional to the template size and not the background graph (i.e., there is no need to maintain detailed statistics on vertices/edges with labels that do not appear in the template). The needed information can be gathered from the background graph following a similar design to the LCC algorithms as it is a simple lookup for each vertex at its own label and at its neighbors’ labels to obtain vertex label and edge label to label information. A reduction step is used to aggregate the results and produce the required distributions for further processing.

D. Using the Estimators

We outline how the computed estimators are used to inform heuristics that improve the performance of our pipeline:

- Constraint Selection and Ordering are done jointly through an iterative greedy algorithm: the algorithm updates the current solution’s (i.e., the currently pruned graph’s) statistics, estimates Effectiveness for all the remaining constraints when run against the current solution, picks the constraint that is the most effective, and uses it for pruning. The cycle finishes when a TDS* constraint that guarantees a precise solution is picked.

- Optimized – Topology-Aware – Individual Constraint Verification. As several walks can be chosen for a specific constraint, it is important to select one that will give good performance. Selecting a walk has no impact on the number of vertices pruned, yet its cost can vary drastically. We attempt to generate all the possible walks and estimate their cost to select the one with the lowest runtime. For large constraints (e.g., constraints containing a lot of cycles) the number of walks generated grows fast, and in this case a greedy algorithm can be used: the walk is built iteratively from the starting vertex and, when several choices are available to add a new edge to the walk, the one with the best cost is chosen. As the walks have to be re-generated each time updated information is available, we chose to use the above greedy algorithm to limit impact on performance.

- The same approach can be used for optimized TDS* constraint verification for enumeration, with the minor difference that only cost needs to be used for ordering.

Deciding when to switch from pruning to direct enumeration allows our pipeline to stop pruning before reaching a precise solution (i.e., guaranteeing no false positives in the pruned graph) and instead switches from graph-pruning directly to enumeration, when an enumeration is the final goal. We can stop pruning when the cost of running a new constraint is higher than the cost of running enumeration. This can happen as enumeration only starts from a vertex with a specific match in the template graph while constraint pruning starts from all of the vertices where the constraint can be applied. One final approach, which we did not implement in our prototype, is to use a hard threshold to stop pruning once all of the remaining constraints have a low estimate for their effectiveness.

VI. PRELIMINARY RESULTS

In this section, we present preliminary results from our prototype shared memory implementation of the pipeline described in §II using the proposed approach for estimating constraint effectiveness (§V-B). We detail our experimental setup in §VI-A, including our testbed and datasets, and verify that our prototype implementation is reasonably optimized by comparing its performance to another state-of-the-art implementation (QFrag [11]). We evaluate the performance improvements enabled by the optimizations, presented in §V-D, in §VI-B and highlight the need to model them. Finally, in §VI-C, we evaluate the pipeline’s performance when the constraints are ordered using: (i) our heuristics informed by constraint effectiveness, (ii) our prior ad-hoc heuristics (§V-A), and (iii) the optimal ordering (identified through a brute-force search). We show that our proposed effectiveness estimate can be used to inform the switch from graph-pruning to early enumeration and achieve significant speedups (§VI-D).

A. Experimental Setup

Pruning Pipeline Implementation. Overheads. To reduce the cost of experimentation, compared to our distributed solution
TABLE I: Performance comparison between QFrag and our pattern matching system. The table shows runtime in seconds, for full enumeration, for QFrag and our pruning-based prototype (PruneJuice-shared). For PruneJuice, we split time-to-solution into pruning (top row) and enumeration (bottom row). We use the same graphs (Patent and YouTube) and query templates (Q4 – Q7) used for evaluation of QFrag in [11]. Note that the other queries used in QFrag’s evaluation are too trivial for PruneJuice (as they have no cycles or repeated labels) and the performance gain is much higher.

<table>
<thead>
<tr>
<th></th>
<th>QFrag</th>
<th>PruneJuice-shared</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Patent</td>
<td>Youtube</td>
</tr>
<tr>
<td>Q4</td>
<td>4.19</td>
<td>8.08</td>
</tr>
<tr>
<td>Q6</td>
<td>5.99</td>
<td>10.26</td>
</tr>
<tr>
<td>Q7</td>
<td>6.36</td>
<td>11.89</td>
</tr>
<tr>
<td>Q8</td>
<td>10.05</td>
<td>14.48</td>
</tr>
</tbody>
</table>

in [1], we developed a prototype shared memory implementation of our pruning pipeline. Better heuristics for this shared-memory prototype are the focus of this paper. To verify that the prototype is optimized at a reasonable level, we have empirically compared (Table I) with recent work (2017) on pattern matching from the database community: QFrag [11]. QFrag runtimes for match enumeration are comparable with the results presented in [11], so we have reasonable confidence that we replicate their experiments well. Our system is considerably faster than QFrag on all experiments ($14 \times$ to $100 \times$). We have communicated with the QFrag authors to make sure we do not misrepresent their work. The magnitude of the difference is not surprising [12] given that QFrag runs on top of Apache Giraph. The caption for Table I provides more details. The processing time also takes into account the overhead to gather statistics to inform the heuristics.

Testbed. The testbed is a large-memory (512GB) machine: 4 socket Intel(R) Xeon(R) CPU E5-2670.v2 @2.50GHz (40 cores). The large memory available enables us to experiment with background graphs that are considerably large.

Datasets. For all graphs (outlined in Table II), we created unirected versions of the graphs; two directed edges are used to represent each undirected edge. We curated the Reddit social-media version of an open archive [13] of billions of public posts and comments from Reddit.com. Reddit allows its users to rate (up/down-vote) others’ posts and comments. The graph has four types of vertices: Author, Post, Comment and Subreddit (a category for Posts). For Posts and Comments there are three possible labels: Positive, Negative, and Neutral (indicating the overall balance of positive and negative votes) or No rating. An edge is possible between an Author and a Post, an Author and a Comment, a Subreddit and a Post, a Post and a Comment (to that post), and between two Comments that have a parent-child relationship. The International Movie Database (IMDB) graph was curated from publicly available data [14]. The graph has five types of vertices: Movie, Genre, Actress, Actor and Director. An edge is only possible between a Movie type vertex and a non-Movie type vertex. We use the smaller Patent and YouTube graphs to compare published results by Serafini et al. [11].

TABLE II: Datasets used for the evaluation. Notation: \( V \) and \( E \) are the set of vertices and edges, respectively.

| Type   | \( |V| \) | \( |E| \) | \( \max_{deg} \) | \( \mu_{deg} \) | \( \sigma_{deg} \) |
|--------|--------|--------|----------------|----------------|----------------|
| Reddit [13] | Real | 2.5B   | 2.7B | 19M | 1.07 | 601.85 |
| IMDB [14] | Real | 5M    | 29M  | 552K | 5.83 | 342.64 |

Search Templates. We use search templates (Fig. 5) based on what we believe to be realistic graph analytics scenarios. The matches are naturally occurring in the background graphs, and are relatively frequent (i.e., we have not artificially modified the background graphs). Finally, to stress the system, the Reddit and IMDB patterns (Fig. 5) include most of the vertex labels in these two graphs. We chose templates that are non-trivial (see previous footnote). The matches are naturally occurring in the background graphs, and are relatively frequent (i.e., we have not artificially modified the background graphs). Finally, to stress the system, the Reddit and IMDB patterns (Fig. 5) include most of the vertex labels in these two graphs. We chose templates that are non-trivial (see previous footnote).

Experimental Methodology. All runtime numbers provided are averages over 10 runs. The runtime does not take into account graph loading. For enumeration, we only consider counting the number of results to avoid I/O effects that, in some cases, became the limiting factor when we tried to store/output the full enumeration results.

B. Impact of Optimizations

To show that incorporating optimizations is key for high performance and that modeling these optimizations is key to accurate cost predictions, we explore the performance impact of the optimizations presented in §V-D.

Figure 6 summarizes the results. Early Termination improves the time to solution by 3 to 5 times. Work Aggregation can fail to produce a meaningful improvement on simple patterns such as IMDB-3; however, it leads to large improvements on complex patterns. Multiple Validation has a smaller impact as it only halves the computation time when vertices can
Fig. 6: The impact of the different optimizations presented in §V-D is shown for the IMDB and the Reddit datasets (described in Table II). The patterns are presented in Fig. 5. All the available constraints are run in the order described §V-D with the topology-aware walk generation. The time to achieve the exact pruned match is normalized by the reference algorithm without optimizations. Baseline runtimes (seconds) are shown on-top-of their respective bars.

actually be validated. This is more probable at the beginning of the NLCC run when most vertices are waiting to be validated.

We note that the benefits of these optimizations are cumulative: in all cases, applying all the optimizations, produces a better result than applying only one, together, optimizations consistently provide $4 \times$ to $10 \times$ performance improvement.

C. Informed Constraint Ordering

This experiment focuses on the impact of using effectiveness for constraint ordering only. To this end, we order the constraints in the set selected by our original approach either based on our previous ad-hoc heuristics (described in §V-A) or based on effectiveness, execute all of them, and report the total runtime (in Fig. 7). The new approach produces an order that leads to better runtime on complex patterns $^5$ such as the Reddit ones and IMDB-1 and -2 and at worst a similar runtime when it is unable to improve upon the intuitive ordering $^6$.

In addition to comparing against our previous ordering solution, another way to estimate the effectiveness of the proposed technique is to compare against the optimal ordering. In this experiment we compare ordering all constraints that can be generated by the estimated effectiveness with an order based on actual effectiveness – computed based on a brute-force search. The order our technique suggests is not always the optimal one as shown in Table III. For IMDB-1, 19 out of 108 constraints were mis-ordered (left column in Table III). A better way to estimate impact, however, is to compare the effectiveness of the constraint which is ranked first by our technique with the effectiveness of the best constraint (right column in Table III). The worst impact is $8.33\%$ over the optimal constraint. Most of the time, however, our solution is near the optimal result and the small mis-ordering does not have a strong impact on performance: two constraints with near similar effectiveness are exchanged.

$^5$In general, the runtimes for IMDB and Reddit are orders of magnitude ($4 \times$ and $3 \times$ respectively) more than for Youtube and Patent.

$^6$In those cases, the intuitive heuristics achieve performance that is close to the optimal due to the rarity of the pattern (IMDB-3), or the label/degree distribution of the background graph (Youtube, Patent). Our proposed approach performs as well as the intuitive heuristics (see Fig. 7 and Fig. 8).

D. Informed Switch to Early Enumeration

When the goal is enumerating all matches, knowing when pruning is no longer necessary and switching to enumeration is of key importance. We attempt to evaluate whether our estimators are accurate enough to forecast when the graph is pruned sufficiently such that it would no longer be useful to run additional pruning constraints. Fig. 8 shows that early enumeration allows us to achieve the result faster than previously by a factor of $1.2 \times$ to $80 \times$ depending on the topology of the graph. Stopping enumeration early is a sensitive approach which is able to find the sweet spot between pruning the graph exactly and working with a non-pruned graph.

VII. RELATED WORK

The volume of prior work on both sequential [10], [15]–[18] and distributed [11], [19]–[24] pattern matching, and more generally on graph processing is humbling. Summaries can be found elsewhere, including in our past work [1]. This section, instead, focuses on how prior work tackles similar optimization decisions:

- **Fixed Heuristics.** Similar to our prior work [1] that uses the ad-hoc heuristics described in §V-A, many other heuristics for pattern matching based on depth-first search have been proposed [15], [16]. However, none of these take the characteristics of the problem at hand (background graph / search template characteristics) into account. In contrast, we propose an approach to dynamically rank constraints depending on the characteristics of the problem solved.

- **Heuristics Informed by Population Information.** Knowledge about the graph topology and label distribution can be
used to enable performance optimizations. For example, several approaches for triangle counting use topology information [25] (specifically, iterating over vertices in order of degree). For pattern matching based on depth-first search, QuickSI [16] uses the frequency of the labels in the graph: prioritizing infrequent labels in the search. Similarly, we use population statistics extracted from the background graph in order to seed our random graph model in §V-C.

Algorithm Selection. Finally, when no single algorithm is optimal, a coarse-grained approach is to select out of a set of algorithms the best one to solve the problem at hand. This involves modeling each algorithm to capture the key elements of the problem space. Kotthoff et al. [26] offer a comprehensive survey focused on combinatorial search problems. As far as we can tell, the problem of algorithm selection for pattern matching in labeled graphs has not been explored (the closest we find is algorithm selection for subgraph isomorphism [27]).

ACKNOWLEDGEMENT
This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA27344. (LLNL-CONF-759624).

REFERENCES

APPENDIX A
RANDOM GRAPH MODEL APPENDIX
Available online at http://www.ece.ubc.ca/~matei/papers/ia3-nicolas-full.pdf