Online Appendix to: Optimizing One-time and Continuous Subgraph Queries using Worst-case Optimal Joins

AMINE MHEDHBI, CHATHURA KANKANAMGE, and SEMIH SALIHOGLU,
University of Waterloo

This online appendix contains: (1) the complexity result of the multiple continuous subgraph query optimization problem; (2) catalogue experimental results; (3) an explanation for how our plans subsume EmptyHeaded plans; (4) comparisons against CFL and TurboFlux; and (5) further continuous subgraph query evaluation experiments.

A COMPLEXITY OF MULTIPLE CONTINUOUS SUBGRAPH QUERY OPTIMIZATION

A.1. Formal Optimization Problem and Its Computational Complexity

Recall that we do not know the exact complexity of the actual computational problem that our optimizer solves. However, we can show that the natural decision version of a slightly more general version of the problem, in which we drop the assumption that the set of DSQs in $\bar{Q}_{DSQ}$ are delta decompositions of a set of subgraph queries is NP-hard. Note that combined plans effectively use common operators across DSQs if the DSQs compute isomorphic sub-queries. Our goal in providing this proof is to make the connection between our optimization problem and the maximum common induced subgraph problem, which is NP-hard. We first define the more general optimization problem we consider:

Definition (Generalized Multiple DSQ Optimization Problem (GDOP)): Given a set of arbitrary delta subgraph queries $\bar{Q}_{DSQ}$, i.e., a set of subgraph queries where one edge is labeled with δ and the other edges with o or n, and an arbitrary full catalogue C, and a target cost k, find whether or not there is a combined plan with cost at most k.

Theorem A.1. *GDOP is NP-hard.*

Proof. We show that the GDOP is NP-hard on instances in which the given a catalogue that has a value of 1 for each cost and selectivity. That is the entries in the catalogue are such that any Q_{k-1} to Q_k extension entry has μ value 1 and the $|A|$ value equal to lists that sum to 1. We call this the *uniform catalogue*. Note that when this is the input catalogue the optimal combined plan is the plan that contains the smallest number of operators. We next show that the **maximum common induced subgraph problem (MCISP)** [40] , which is NP-hard, reduces to GDOP. Given two graphs G_1 and G_2 and a target value t, MCISP is the problem of finding whether or not there is a subgraph H with at least t vertices that is an induced subgraph of both G_1 and G_2, i.e., the projection of G_1 and G_2 onto the vertices in H gives H.

The reduction is as follows. Take an instance of MCISP with graphs G_1 and G_2 and a target induced subgraph of size t. Assume the nodes in G_1 are labeled with $a_1, a_2, \ldots, a_{m_1}$ and each node in G_2 is labeled with $b_1, b_2, \ldots, b_{m_2}$. We first construct an instance of GDOP as follows. Label each edge of G_1 and G_2 with o, and extend both G_1 and G_2 with a new edge $x \rightarrow y$ with label δ and connect

0362-5915/2021/05-ART6 $15.00
https://doi.org/10.1145/3446980

both x and y to each node in G_1 and G_2. These labeled and extended G_1 and G_2 graphs are now delta subgraph queries, and we refer to them as DSQ_1 and DSQ_2. Now consider solving GDOP on DSQ_1 and DSQ_2 using a uniform catalogue and a target cost of $m_1 + m_2 + 1 - t$, which, due to the structure of the catalogue, is the problem of finding a combined plan with at most $m_1 + m_2 + 1 - t$ operators.

First, observe that any combined plan needs to have exactly two sink E/I operators because there are two DSQs, i.e., in the DAG of any correct combined plan for this GDOP instances there will be two final "branches" leading to sink operators. Second, observe that we only need to consider combined plans whose DAGs have the following structure: (1) start with a source Scan that matches $\xrightarrow{\delta}$ as usual; (2) a chain of z E/I operators each giving its output to one output E/I operator, which compute a common sub-query for both DSQ_1 and DSQ_2, where the last E/I operator gives its output to two operators (to start the final two "branches"); (3) two branches, one with $r_1 = m_1 - z$ many and the other with $r_2 = m_2 - z$ many E/I operators, each giving its output to one output E/I operator. Any combined plan that branches and merges multiple times is suboptimal, because, we can always keep the last two branches and then only keep one chain back to the source Scan operator, so remove all but one of the previous branches that eventually merge, which strictly decreases the number of operators in the combined plan. Therefore any combined plan effectively starts with a Scan operator that evaluates the extra $x \rightarrow y$ edge we added to G_1 and G_2 and then a z-size common induced subgraph of G_1 and G_2 and then in two separate branches of E/I operators evaluates the rest of the vertices in G_1 and G_2, with a cost of $m_1 + m_2 + 1 - z$ (+1 is for the initial scan operator). Therefore, there is an induced subgraph of size at least t if and only if there is a combined plan in the GDOP instance with at most $m_1 + m_2 + 1 - t$ cost, completing the proof. □

B CATALOGUE EXPERIMENTS

We present preliminary experiments to show two tradeoffs: (1) the space vs. estimation quality tradeoff that parameter h determines; and (2) construction time vs. estimation quality tradeoff that parameter z determines. For estimation quality we evaluate cardinality estimation and omit the estimation of adjacency list sizes, i.e., the $|A|$ column, that we use in our i-cost estimates. We first generated all 5-vertex size unlabeled queries. This gives us 535 queries. For each query, we assign labels at random given the number of labels in the dataset (we consider Amazon with 1 label, Google with 3 labels). Then for each dataset, we construct two sets of catalogues: (1) we fix z to 1,000, and construct a catalogue with $h = 2$, $h = 3$, and $h = 4$ and record the number of entries in the catalogue; (2) we fix h to 3 and construct a catalogue with $z = 100$, $z = 500$, $z = 1{,}000$, and $z = 5{,}000$ and record the construction time. Then, for each labeled query Q, we first compute its actual cardinality, $|Q_{true}|$, and record the estimated cardinality of Q, Q_{est} for each catalogue we constructed. Using these estimation we record the q-error of the estimation, which is $\max(|Q_{est}| / |Q_{true}|, |Q_{true}| / |Q_{est}|)$. This is an error metric used in prior cardinality estimation work [35] that is at least 1, where 1 indicates completely accurate estimation. As a very basic baseline, we also compared our catalogues to the cardinality estimator of PostgreSQL. For each dataset, we created an Edge relation E(from, to). We create two composite indexes on the table on (from, to) and (to, from) which are equivalent to our forward and backward adjacency lists. We collected stats on each table through the ANALYZE command. We obtain PostgreSQL's estimate by writing each query in an equivalent SQL select-join query and running EXPLAIN on the SQL query.

Our results are shown in Tables 1 and 2 as cumulative distributions as follows: for different q-error bounds τ, we show the number of queries that a particular catalogue estimated with q-error at most τ. As expected, larger h and larger z values lead to less q-error, while respectively yielding

Table 1. Q-error and Catalogue Creation Time (CT) in Secs for GraphflowDB for Different z Values

	z	CT	≤2	≤3	≤3	≤5	≤10	>20
Am	100	0.1	318	445	510	526	529	535
	500	0.3	384	486	520	527	530	535
	1,000	0.5	383	481	519	529	532	535
	5,000	1.5	384	475	518	529	532	535
Go_3	100	3.1	166	276	356	415	561	535
	500	9.3	214	310	371	430	477	535
	1,000	17.0	222	315	371	430	475	535
	5,000	66.1	219	322	373	432	473	535

Table 2. Postgres (PG) and GraphflowDB (GF) Q-error and Number of Catalogue Entries (|R|) for GF for Different h Values

		h	|R|	≤2	≤3	≤3	≤5	≤10	>20
Am	GF	2	8	348	464	512	523	527	535
		3	138	381	482	512	524	527	535
		4	2858	498	510	518	524	527	535
	PG	–	–	15	15	23	23	25	535
Go_3	GF	2	144	181	289	375	447	492	535
		3	20.3K	222	315	371	430	475	535
		4	11.9M	441	497	515	524	529	535
	PG	–	–	0	0	0	0	0	535

larger catalogue sizes and longer construction times. The biggest q-error differences are obtained when moving from $h = 3$ to $h = 4$ and $z = 100$ to $z = 500$. There are a few exception τ values when the larger h or z values lead to very minor decreases in the number of queries within the τ bound but the trend holds broadly.

C SUBSUMED EMPTYHEADED PLANS

We show that our plan space contains EmptyHeaded's GHD-based plans that satisfy the projection constraint. For details on GHDs and how EmptyHeaded picks GHDs we refer the reader to Reference [2]. Briefly, a GHD D of Q is a decomposition of Q where each node i is labelled with a subquery Q_i of Q. The interpretation of a GHD D as a join plan is as follows: each subquery is evaluated using Generic Join first and materialized into an intermediate table. Then, starting from the leaves, each table is joined into its parent in an arbitrary order. So a GHD can easily be turned into a join plan T in our notation (from Section 3.2) by "expanding" each sub-query Q_i into a WCO subplan according to the chosen QVO that EmptyHeaded picks for Q_i and adding intermediate nodes in T that are the results of the joins that EmptyHeaded performs. Given Q, EmptyHeaded picks the GHD D^* for Q as follows. First, EmptyHeaded loops over each GHD D of Q, and computes the worst-case size of the subqueries, which are computed by the AGM bounds of these queries (i.e., the minimum *fractional edge covers* of sub-queries; see Reference [8]). The maximum size of the subqueries is the width of GHD and the GHD with the minimum width is picked. This effectively implies that one of these GHDs satisfy our projection constraint. This is because adding a

missing query edge to $Q_i(V_i, E_i)$ can only decrease its fractional edge cover. To see this consider $Q'_i(V_i, E'_i)$, which contains V' but also any missing query edge in E_i. Any fractional edge cover for Q_i is a fractional edge cover for Q'_i (by giving weight 0 to $E'_i - E_i$ in the cover), so the minimum fractional edge cover of Q'_i is at most that for Q_i, proving that D^* is in our plan space.

We verified that for every query from Figure 10, the plans EmptyHeaded picks satisfy the projection constraint. However, there are minimum-width GHDs that do not satisfy this constraint. For example, for Q10, EmptyHeaded finds two minimum-width GHDs: (i) one that joins a diamond and a triangle (width 2) and (ii) one that joins a three path ($a_2a_1a_3a_4$) joined with a triangle with an extended edge (also width 2). The first GHD satisfies the projection constraint, while the second one does not. EmptyHeaded (arbitrarily) picks the first GHD. As we argued in Section 3.2.1 , satisfying the projection constraint is not a disadvantage, as it makes the plans generate fewer intermediate tuples. For example, on a Gnutella peer-to-peer graph [37] (neither GHD finished in a reasonable amount of time on our datasets from Table 9), the first GHD for Q10 takes around 150ms, while the second one does not finish within 30 minutes.

D CFL COMPARISON

CFL [10] is one of the state-of-the-art subgraph matching algorithms whose code is available. The algorithm can evaluate labelled subgraph queries as in our setting. The main optimization of CFL is what is referred to as "postponing Cartesian products" in the query. These are conditionally independent parts of the query that can be matched separately and appear as Cartesian products in the output. CFL decomposes a query into a dense *core* and a *forest*. Broadly, the algorithm first matches the core, where fewer matches are expected and there is less chance of independence between the parts. Then the forest is matched. In both parts, any detected Cartesian products are postponed and evaluated independently. This reduces the number of intermediate results the algorithm generates. CFL also builds an index called CPI, which is used to quickly enumerate matches of paths in the query during evaluation. We follow the setting from the evaluation section of Reference [10]. We obtained the CFL code and 6 different query sets used in Reference [10] from the authors. Each query set contains 100 randomly generated queries that are either sparse (average query vertex degree ≤ 3) or dense (average query vertex degree > 3). We used three sparse query sets Q10s, Q15, and Q20s containing queries with 10, 15, and 20 query vertices, respectively. Similarly, we used three dense query sets Q10d, Q15d, and Q20d. To be close to their setup, we use the `human` dataset from the original CFL paper. The dataset contains 86,282 edges, 4,674 vertices, 44 distinct labels. We report the average runtime per query for each query set when we limit the output to 10^5 and 10^8 matches as done in Reference [10]. Table 3 compares the runtime of GraphflowDB and CFL on the 6 query sets. Except for one of our experiments, on Q10d with 10^5 output size limit, GraphflowDB's runtimes are faster (between 1.2× to 12.2×) than CFL. We note that although our runtime results are faster than CFL on average, readers should not interpret these results as one approach being superior to another. For example, we think the postponing

Table 3. Average Runtime (seconds) of GraphflowDB (GF) and CFL on Large Queries

|T|		Q10s	Q15s	Q20s	Q10d	Q15d	Q20d
10^5	GF	7.3	6.0	5.5	29.2(**2.2x**)	99.8	142.0
	CFL	9.3(**1.2x**)	17.5(**2.9x**)	40.5(**7.3x**)	13.2	389.9(**3.9x**)	1,140.7(**8.0x**)
10^8	GF	625.6	665.5	797.2	1,159.6	1,906.2	1,556.9
	CFL	4,818.9(**7.7x**)	5,898.1(**8.8x**)	7,104.1(**8.9x**)	7,974.3(**6.8x**)	11,656.2(**6.1x**)	19,135.7(**12.2x**)

Q*i(s/d)* is a query set of 100 randomly generated queries where *i* is the number of vertices and *s* and *d* specify sparse and dense queries, respectively as specified in Appendix D.

Table 4. Turboflux (T_f) vs. GraphflowDB (GF) on Continuous Subgraph Queries Diamond (Q_D), Diamond-X (Q_{DX}), 4-Clique (Q_{4C}), and 5-Clique (Q_{5C}) from the Query Set SEED

		Q_D	Q_{4C}	Q_{5C}
Am	GF	3.5	2.1	9.1
	T_f	11.4 (**3.3x**)	13.2 (**6.3x**)	1069 (**117.5x**)
Go	GF	9.1	3.8	50.4
	T_f	29.9 (**3.3x**)	41.6 (**10.9x**)	3090 (**61.3x**)

of Cartesian products optimization and a CPI index are good techniques and can improve our approach. However, one major advantage of our approach is that we do flat tuple-based processing using standard database operators, so our techniques can easily be integrated into existing graph databases. It is less clear how to decompose CFL-style processing into database operators.

E TURBOFLUX COMPARISON

TurboFlux [30] evaluates a query using a *data centric graph* (DCG), which is a compressed representation of partial matches of the query in $G(E_G, V_G)$. Briefly, the DCG is a multigraph $G_{DCG}(V_{DCG}, E_{DCG})$ where $V_{DCG} = V_G$ and E_{DCG} contains $|V_Q| - 1$ parallel edges for each $e \in E_G$. Each parallel edge has a *state* of *null* (N), *implicit* (IM), or *explicit* (EX) and a *label* which is one of the query vertices in Q. An EX edge $u \rightarrow v$ with label $a_i \in V_Q$ indicate a set of successful matches of vertices (according to an order) to query vertices where v matches a_i. Updates to G, TurboFlux transitions the states of the edges and then runs a subgraph matching algorithm on the DCG.

We obtained the code from the original authors. We used four different continuous subgraph queries from the SEED query set: (i) a Diamond (Q_D); (ii) a Diamond-X (Q_{DX}); (iii) a 4-Clique (Q_{4C}); and (iv) a 5-Clique (Q_{5C}). As in previous experiments we pre-loaded both TurboFlux and GraphflowDB with a random 90% of the dataset. We streamed in the remaining 10% of edges one edge at a time because TurboFlux does not support batching of updates. We show the results of this experiment in Table 4. Across all datasets and queries, GraphflowDB outperforms TurboFlux by at least 3.3× and by up to 117.5×. As we emphasized in our one-time query CFL comparisons, readers should not conclude from these experiments that our approach is superior to TurboFlux's approach. The compared approaches and the actual implementations of these approaches are very different (and we were only provided the binary). We provide these experiments merely for completeness of our work and sanity checks to verify that our implementation is competitive with existing recent solutions from literature. We believe approaches such as DCG that allow compressed representations are good techniques. One important distinction to note is that our approach was specifically designed to be easily integrated into a GDBMS and our implementation is part of a GDBMS architecture. In contrast it is less clear how to decompose TurboFlux-style processing into actual database implementations. This is an interesting research direction.

F RUNTIME AND EXECUTION METRICS FOR CONTINUOUS SUBGRAPH QUERIES

Table 5 reports the rest of our experiments from Section 6.3 on Epinions and Google datasets.

Table 5. Runtime (Seconds) of B_{ns}, B_s, Gr, and G_{opis} on 4Cs, 4Cs5C, SEED, and MagicRec Query Sets

		Runtime	**I-cost**	**Runtime**	**I-cost**	**Runtime**	**I-cost**
		$4Cs_1$		**$4Cs_2$**		**$4Cs_3$**	
Ep	B_{ns}	5.51	1.488B (**84%**)	0.56	0.163B (**65%**)	0.33	0.052B (**50%**)
	B_s	5.15 (**1.07x**)	1.347B (**1.10x**)	0.43 (**1.30x**)	0.125B (**1.30x**)	0.21 (**1.57x**)	0.035B (**1.49x**)
	Gr	4.95 (**1.11x**)	1.328B (**1.20x**)	0.49 (**1.14x**)	0.126B (**1.30x**)	0.22 (**1.50x**)	0.037B (**1.41x**)
	Gr_p	4.78 (**1.15x**)	1.244B (**1.20x**)	0.49 (**1.14x**)	0.126B (**1.30x**)	0.18 (**1.83x**)	0.030B (**1.73x**)
Go	B_{ns}	14.61	3.032B (**58%**)	2.70	0.515B (**38%**)	1.31	0.188B (**39%**)
	B_s	13.03 (**1.12x**)	2.226B (**1.36x**)	2.14 (**1.26x**)	0.314B (**1.64x**)	1.12 (**1.17x**)	0.126B (**1.49x**)
	Gr	13.10 (**1.12x**)	2.073B (**1.46x**)	2.23 (**1.21x**)	0.285B (**1.81x**)	1.00 (**1.31x**)	0.118B (**1.59x**)
	Gr_p	13.16 (**1.11x**)	1.864B (**1.63x**)	1.99 (**1.36x**)	0.278B (**1.85x**)	0.94 (**1.39x**)	0.106B (**1.77x**)
		$4Cs5C_1$		**$4Cs5C_2$**		**$4Cs5C_3$**	
Ep	B_{ns}	16.71	5.607B (**64%**)	0.97	0.278B (**22%**)	0.42	0.074B (**7%**)
	B_s	16.35 (**1.02x**)	5.339B (**1.05x**)	0.95 (**1.02x**)	0.218B (**1.28x**)	0.29 (**1.45x**)	0.048B (**1.54x**)
	Gr	15.78 (**1.06x**)	5.320B (**1.05x**)	0.82 (**1.18x**)	0.215B (**1.29x**)	0.36 (**1.17x**)	0.048B (**1.54x**)
	Gr_p	15.80 (**1.06x**)	5.235B (**1.07x**)	0.66 (**1.47x**)	0.209B (**1.33x**)	0.27 (**1.56x**)	0.041B (**1.80x**)
Go	B_{ns}	33.96	5.422B (**36%**)	3.54	0.607B (**6%**)	1.63	0.211B (**1%**)
	B_s	33.1 (**1.03x**)	4.436B (**1.22x**)	2.88 (**1.23x**)	0.374B (**1.63x**)	0.99 (**1.65x**)	0.136B (**1.55x**)
	Gr	32.3 (**1.05x**)	4.283B (**1.27x**)	2.64 (**1.34x**)	0.343B (**1.77x**)	0.97 (**1.68x**)	0.126B (**1.67x**)
	Gr_p	32.8 (**1.04x**)	4.073B (**1.33x**)	2.71 (**1.31x**)	0.332B (**1.83x**)	0.97 (**1.68x**)	0.113B (**1.87x**)
		$SEED_1$		**$SEED_2$**		**$SEED_3$**	
Ep	B_{ns}	205.9	47.76B (**86%**)	8.45	2.071B (**61%**)	2.43	0.430B (**43%**)
	B_s	197.8 (**1.04x**)	46.03B (**1.04x**)	7.68 (**1.10x**)	1.740B (**1.19x**)	1.80 (**1.35x**)	0.323B (**1.33x**)
	Gr	197.8 (**1.04x**)	46.03B (**1.04x**)	7.68 (**1.10x**)	1.732B (**1.20x**)	1.81 (**1.34x**)	0.317B (**1.36x**)
	Gr_p	192.8 (**1.07x**)	44.75B (**1.07x**)	7.45 (**1.13x**)	1.691B (**1.22x**)	1.81 (**1.34x**)	0.311B (**1.38x**)
Go	B_{ns}	97.9	13.04B (**74%**)	7.25	0.715B (**34%**)	3.31	0.208B (**28%**)
	B_s	97.4 (**1.01**)	11.98B (**1.09x**)	5.50 (**1.32x**)	0.499B (**1.43x**)	2.37 (**1.40x**)	0.146B (**1.42x**)
	Gr	81.6 (**1.20x**)	11.51B (**1.13x**)	5.64 (**1.29x**)	0.500B (**1.43x**)	1.77 (**1.87x**)	0.114B (**1.82x**)
	Gr_p	81.6 (**1.20x**)	11.51B (**1.13x**)	5.31 (**1.37x**)	0.474B (**1.51x**)	1.61 (**2.06x**)	0.109B (**1.91x**)
		$MagicRec_1$		$MagicRec_2$		$MagicRec_3$	
Ep	B_{ns}	1524	40.64B (**19%**)	40.2	9.999B (**28%**)	7.66	2.048B (**24%**)
	B_s	1416 (**1.08x**)	18.06B (**2.25x**)	24.6 (**1.63x**)	5.414B (**1.85x**)	3.73 (**2.05x**)	1.014B (**2.02x**)
	Gr	1414 (**1.08x**)	17.65B (**2.30x**)	24.4 (**1.65x**)	5.121B (**1.95x**)	3.66 (**2.09x**)	0.929B (**2.20x**)
Go	B_{ns}	475.3	43.77B (**20%**)	18.2	7.345B (**26%**)	5.48	1.600B (**23%**)
	B_s	430.4 (**1.10x**)	20.19B (**2.17x**)	10.3 (**1.77x**)	3.922B (**1.87x**)	2.78 (**1.97x**)	0.786B (**2.04x**)
	Gr	427.2 (**1.11x**)	19.88B (**2.20x**)	9.9 (**1.84x**)	3.607B (**2.04x**)	2.62 (**2.09x**)	0.702B (**2.23x**)

The percentage value next to B_{ns} total i-cost shows the percentage of work done in the last level. Values in parentheses show the factor of improvement of the runtime over B_{ns}.