Measuring the Width of a Join Graph for (Graph) Queries Project Specification

Zeyuan Hu
CS386D Spring 2022

April 15, 2022

1 Introduction

In this project, we are interested in measuring two topological features of graph queries in benchmarks called *treewidth* and *hypertree width*. In this section, we provide necessary background to help you get started with the project. We provide some references if you are interested in learning more. More references can be found on the term project web page.

1.1 Relational

Before talking about graph queries, we first present results based on relational model. In general, query evaluation problem for relational algebra is PSPACE-complete [Kol16]. In other words, `SQL` evaluation is PSPACE-complete. Given the hardness of the problem, people have been focusing on `SELECT .. FROM .. WHERE` block from `SQL`, which is called *conjunctive queries (CQs)* (i.e., k-way join queries) in the literature. It turns out that query evaluation for CQs is NP-complete [CM77]. However, some CQs are easier to evaluate than others. For example, `SELECT 1` is easier than `SELECT * FROM R,S,T WHERE R.b = S.b AND S.c = T.c AND T.a = R.a`. A natural question to ask is what classes of queries are tractable, i.e., can be evaluated in polynomial time. To answer this question, people have been focusing on classifying graph representation of CQs based on some intrinsic characteristics of graphs. If a graph of one query Q_1 is "nicer" than a graph of another query Q_2, then Q_1 is "easier" to evaluate than Q_2. Such graph is called *join tree* $\mathcal{T}$ [1]. A class of queries that have $\mathcal{T}$ is called *acyclic CQs*; otherwise, is called *cyclic CQs*. Acyclic CQs can be evaluated in polynomial time using Yannakakis's algorithm [Yan81].

Let us revisit the definition of *query graph* (i.e., *factor graph*, *dual constraint graph*) first before presenting $\mathcal{T}$.

Definition 1 (Query Graph)**.** *$\mathcal{G} = (V, E)$ is a query graph if $v \in V$ represents a relation from Q and $(v_1, v_2) \in E$ if $attr(v_1) \cap attr(v_2) \neq \emptyset$.*

$attr(R)$ is a function that extracts attributes associated with the input (e.g., if input is a relation R, $attr$ extracts attributes associated with R.) Note that $\mathcal{G}$ can contain cycles. For example, for $U(c) \bowtie N(c, a, d) \bowtie E(c, a)$, $\mathcal{G}$ is shown in Figure 1.

It turns out, sometimes, even when the query graph does not look like a tree, it is in fact a tree, if some of its edges are redundant and can be removed, leaving behind a tree structure. An edge is *redundant* if its removal from $\mathcal{G}$ does not change the query result. For $\mathcal{G}$, like Figure 1, there are multiple ways (i.e., 3 ways) to remove edges without changing query result. However, from query evaluation efficiency perspective (e.g., the size of intermediate result), removal one edge is better than the other. This is where join tree comes from; it specifies what kind of tree we are looking for, i.e., gives efficient evaluation.

Definition 2 (Connectedness Property [Dec03])**.** *An arc subgraph of the query graph satisfies the connectedness property if and only if for each two nodes that share a variable, there is at least one path between those two nodes such that each node on the path containing the shared variables.*

[1] Please don't confuse with join tree under the context of query plan structure (e.g., left-deep join tree) [GMUW08]. This is an unfortunate coincidence.

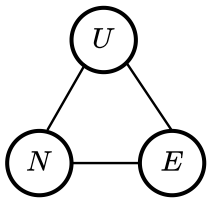

Figure 1: $\mathcal{G}$ for $U \bowtie N \bowtie E$.

For example, Figure 2 shows an example of connectedness property violation. There is only one path between N and E but U on the path doesn't contain attribute a.

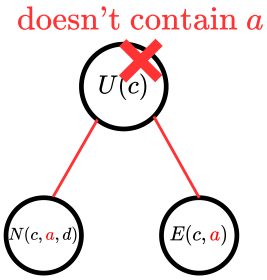

Figure 2: Removing (N, E) violates connectedness property because U doesn't contain attribute a

Definition 3 (Join Tree [Dec03])**.** *An arc subgraph of the query graph that satisfies the connectedness property is called a join graph. A join graph that is a tree is called a join tree.*

Figure 3 shows a join tree from $U(c) \bowtie N(c, a, d) \bowtie E(c, a)$.

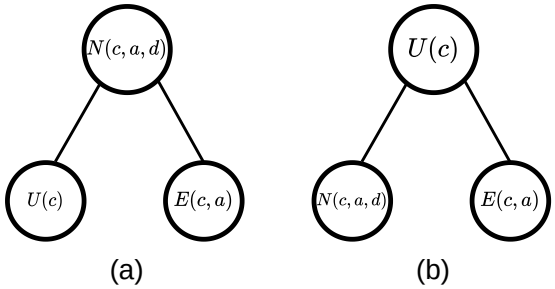

Figure 3: (a) is a join tree but (b) is not.

There are many ways to find a join tree (see [BFMY83] for a list). The most common one is the GYO algorithm [GMUW08]. As mentioned earlier, queries that have join trees are called acyclic queries and acyclic queries can be evaluated in polynomial time. However, we are not stating queries that can be evaluated in polynomial time are acyclic queries. In fact, part of cyclic queries can be evaluated in polynomial time as well. As an example, triangle queries $R(a, b) \bowtie S(b, c) \bowtie T(c, a)$ can be evaluated in polynomial time. Research has shown that queries with *bounded treewidth* can be evaluated in polynomial time [GSS01], which will be the focus of this project. To define treewidth, we need to present a few concepts first.

Definition 4 (Hypergraph [GMUW08])**.** *Suppose we have k relations $R_1, \ldots, R_k$ in the Q. Let $D = attr(R_1) \cup \cdots \cup attr(R_k)$ be the set of attributes appearing in those k relations. $H = (V(H), E(H))$ is a hypergraph for Q if each $v \in V(H)$ represents some attribute from D and $e \in E(H)$ corresponds to some relation R_i. Note that $e \subseteq V$. e is called a hyperedge.*

For example, H for $R(a, b, c) \bowtie S(c, d)$ is Figure 4.

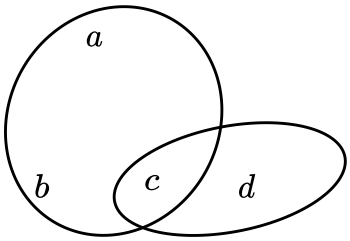

Figure 4: H for $R(a,b,c) \bowtie S(c,d)$.

To handle cyclic queries, one idea is to transform their query graphs into a join tree so that we can employ Yannakakis's algorithm. To do so, we use a technique called *tree decomposition*. Let us first introduce what the result of *tree decomposition* look like, which is also called *tree decomposition*!

Definition 5 (Tree Decomposition [GM14])**.** *A tree decomposition of a hypergraph H is a tuple $(T, (B_t)_{t \in V(T)})$, where T is a tree and $(B_t)_{t \in V(T)}$ is a family of subsets of $V(H)$ such that*

1. *for each $e \in E(H)$ there is a node $t \in V(T)$ such that $e \subseteq B_t$; and*
2. *for each $v \in V(H)$ the set $\{t \in V(T) \mid v \in B_t\}$ is connected in T (i.e., connectedness property as in Definition 2).*

The sets B_t are called the bags of the decomposition.

We first observe that join tree is a tree decomposition. Consider $\mathcal{T}$ in Figure 3 (a) as an example. Let us label the nodes in $\mathcal{T}$ as shown in Figure 5. Then, $B_1 = \{c, a\}$, $B_2 = \{c\}$, and $B_3 = \{c, a\}$. $V(H) = \{a, b, c\}$. Following Definition 5, Figure 5 is a tree decomposition.

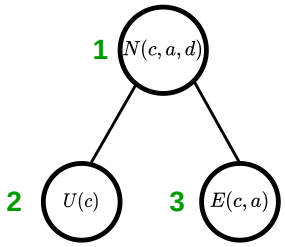

Figure 5: A tree decomposition of $U(c) \bowtie N(c,a,d) \bowtie E(c,a)$.

For another example, consider $R(a,b) \bowtie S(b,c) \bowtie T(c,a)$. A tree decomposition of the query is shown in Figure 6.

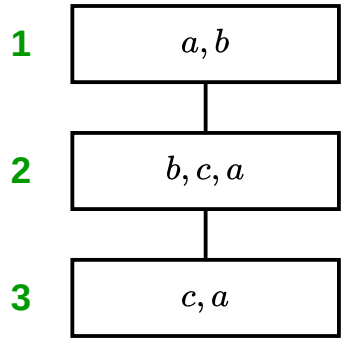

Figure 6: A tree decomposition of $R(a,b) \bowtie S(b,c) \bowtie T(c,a)$.

Definition 6 (Width, TreeWidth [GM14]). *The width of a tree decomposition* $(T, (B_t)_{t \in V(T)})$ *is* $\max\{|B_t| \mid t \in V(t)\}$. *The treewidth* $tw(H)$ *of a hypergraph* H *is the minimum of the widths of all tree decompositions of* H. *Expressed in equation, we have*

$$tw(H) \coloneqq \min_{(T,(B_t)_{t \in V(T)})} \max_{t \in V(t)} |B_t| - 1 \tag{1}$$

For example, for Figure 6, the treewidth is 2 because all tree decompositions of $H = (\{a, b, c\}, \{(a, b), (b, c), (c, a)\})$ will have some node that contains three attributes, just like Figure 6.

To conclude treewidth introduction, we finish the whole story of tractable query evaluation that starts our journey on this route.

The general idea for queries that are tractable is we first represent queries in hypergraph H and run tree decomposition on it. If $tw(H)$ is *bounded* meaning $tw(H)$ value exists, then we can evaluate such queries in polynomial time: bounded tree width implies that there exists a tree decomposition $(T, (B_t)_{t \in V(T)})$ such that the size of bags $|B_t|$ $(t \in V(T))$ in this tree decomposition is bounded. Then, we can evaluate relations associated with each bag and materialized those results. Then, run Yannakakis's algorithm on T with each bag being a relation.

To find class of queries that are tractable, we can, for each query we see, we check whether it has bounded treewidth $tw(H)$, if so, we put its hypergraph into a class $\mathcal{H}$. Then, we can say if $\mathcal{H}$ has bounded tree width, $\mathcal{Q}(\mathcal{H})$ is tractable.

Definition 7 (bounded tree width [GM14]). *For a class of* $\mathcal{H}$ *of hypergraphs, let* $\mathcal{Q}(\mathcal{H})$ *be the class of all queries whose hypergraph is contained in* $\mathcal{H}$, *we say that a class* $\mathcal{H}$ *of hypergraphs is of bounded tree width if there is a* k *such that* $tw(H) \leq k$ *for all* $H \in \mathcal{H}$.

Theorem 1 ([GSS01, GM14]). *If* $\mathcal{H}$ *is of bounded tree width,* $\mathcal{Q}(\mathcal{H})$ *can be evaluated in polynomial time. In fact, this result also holds for* $\mathcal{H}$ *of bounded hyperegde size, i.e., a class* $\mathcal{H}$ *for which* $\max\{|e| \mid \exists H = (V, E) \in \mathcal{H} : e \in E\}$ *exists (equivalently, the edge size in* $\mathcal{H}$ *is bounded by a constant* r*).*

In fact, when the schema of the database is considered as fixed (i.e., arity of each relation in the database is bounded by a constant [Kou22]), $\mathcal{H}$ with bounded treewidth is the only polynomial-time solvable class [GSS01, Mar13]. However, research has shown that there are other decomposition methods that are able to identify even larger tractable classes. Nevertheless, measuring whether treewidth is bounded is still interesting. One one hand, latest query evaluation algorithm relies on treewidth [KNRR16]. On the other hand, despite recent research [FGLP21] on measuring other width parameters associated with other decomposition methods (e.g., $hw(H)$ for *hypertree width* [2], $ghw(H)$ for *generalized hypertree width*), bounded $tw(H)$ leads to a stronger claim on the properties of queries in the benchmarks. Note that $ghw(H) \leq tw(H) + 1$ and it is possible that $tw(H)$ is unbounded given $ghw(H)$ is bounded [GM14]. In addition, $ghw(H) \leq hw(H) \leq 3 \cdot ghw(H) + 1$ [AGG07]. The takeaway is that bounded $ghw(H)$ or $hw(H)$ does not imply the $tw(H)$ is bounded (but, bounded $tw(H)$ entails bounded $hw(H)$ [GGS14]). In other words, if benchmark queries show that majority of queries have bounded $tw(H)$, the queries are much "nicer" than what have been showing in the existing literature.

1.2 Graph

There is a great similarity between graph queries and relational queries. Like conjunctive queries for relational queries, the focus of study in graph queries is on *conjunctive regular path queries with inverses (C2RPQs)* [BRV16].

Example 1. *Consider a graph database* $\mathcal{G} = (N, E)$ *of researchers, papers, conferences, and journals over the alphabet* $\Sigma = \{creator, inJournal, inConf\}$. *The set of edges* E *consists of the following:*

1. *All tuples* $(r, creator, p)$ *such that* r *is a researcher that (co-)authors paper* p;
2. *Each tuple* $(p, inJournal, j)$ *such that paper* p *appeared in journal* j; *and*

[2] See Section 1.3.2 for details.

3. All tuples $(p, inConf, c)$ such that paper p was published in conference c

A C2RPQ $\gamma(x, y)$ is defined as

$$\gamma(x, y) = \exists z \exists w((x, creator, z) \wedge (z, inConf, w) \wedge (z, creator^{-}, y)) \tag{2}$$

, which finds all the co-authors of a conference paper.

To use treewidth as a parameter for tractable C2RPQs, we need to define query graph (1) accordingly for C2RPQs. This is not hard to do: each $(u_i, \mathcal{A}, v_i)$ can be translated into $\mathcal{A}(u_i, v_i)$, a relation with two attributes. In (2), we effectively have,

$$\gamma(x, y) = \text{creator}(x, z) \bowtie \text{inConf}(z, w) \bowtie \text{creator}^{-}(z, y) \tag{3}$$

a 3-way join. How each newly-formed relation is computed is not a concern of our project because treewidth of a query is a syntactic feature. But if you're interested, please reference [MW95]. With this translation, we can apply the tree decomposition technique from 1.1 and find our bounded treewidth for benchmark queries.

1.3 Extra Credits

Note that tree has treewidth 1 [GGS14]. However, this does not mean $tw(H) = 1$. For example, the tree in Figure 5 formed by U, N, E three nodes (ignore the attributes) has treewidth 1 because its tree decomposition, shown in Figure 7, has width 1. However, $tw(H) = 2$ because, by definition, $e = \{c, a, d\} \subseteq B_t$ for some t. Thus, the width of a tree decomposition is at least $|\{c, a, d\}| - 1 = 2$. In fact, Figure 5 is one such tree decomposition that reaches $tw(H) = 2$.

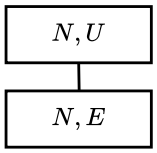

Figure 7: A tree decomposition of the tree consists of N, U, E nodes in Figure 5.

This example illustrates a problem with treewidth as a measurement: Acyclic CQs can have arbitrary treewidth [3]. For H representing $U(c) \bowtie N(c, a, d) \bowtie E(c, a)$, $tw(H) = 2$. If we replace $N(c, a, d)$ with $N(c, a, d, e)$ and keep the rest unchanged, $tw(H) = 3$. In general, we can keep adding attributes to N while maintaining connectedness property, we can have $tw(H) \to \infty$. However, $U(c) \bowtie N \bowtie E$ will always be an acyclic CQ. Put it differently, treewidth is not a good parameter to capture the degree of acyclicity of hypergraph (i.e., good parameter needs to meet the three conditions (requirements) listed in §1.3 [GGLS16]), despite from the tractability perspective, it already captures the only polynomial-solvable class when the schema of the database is fixed [4].

Thus, we want to find a width parameter such that acyclic CQs will have the same width, preferably 1, no matter the *arity* (i.e., the number of attributes) of relation(s) in queries. *hypertree width* and its associated *hypertree decomposition* meet this criterion.

[3] Additional implication is that a query with unbounded treewidth does not necessarily imply that such query is intractable, which makes treewidth, a possible indicator for query evaluation tractability, look problematic.

[4] Hypertree width from Section 1.3.2 is equivalent to treewidth in terms of representing tractable class when the database schema is fixed. However, hypertree width is able to capture larger tractable class considering database schema is not fixed and be better representative of the degree of acyclicity (i.e., captures Yannakakis's algorithm), which make hypertree width preferable than treewidth.

1.3.1 Part 1

This part concerns the intuition behind hypertree width and hypertree decomposition. You don't need to understand those two terms to earn extra credits associated with this part (You do in Section 1.3.2). This part takes **20%** of project full marks.

Continue our example regarding $U(c) \bowtie N(c, a, d) \bowtie E(c, a)$. We observe that the tree decomposition in Figure 7 always has width 1 no matter how we change attributes of N. Figure 7 represents a tree decomposition of a hypergraph $H' = (\{N, E, U\}, \{(N, U), (N, E)\})$, which itself is a tree decomposition obtained from H representing $U(c) \bowtie N(c, a, d) \bowtie E(c, a)$. The goal of this part is to measure $tw(H')$. Intuitively, we change what treewidth represents: instead of measuring the treewidth in terms of number of attributes within each bag of $(T, (B_t)_{t \in V(T)})$ (from H), now we measure the number of hyperedges, i.e., relations, contained in each bag of $(T', (B_t)_{t \in V(T')})$ (from H'). We will exploit this intuition in Section 1.3.2.

Instruction. After using your tree decomposition algorithm to figure out the $tw(H)$, construct H' and re-run your algorithm on H' and report $tw(H')$. In addition, compare $tw(H)$ with $tw(H')$ and discuss your findings, e.g., pick one query and show its $(T, (B_t)_{t \in V(T)})$ and $(T', (B_t)_{t \in V(T')})$ and note your observations.

1.3.2 Part 2

In this part, you will need to perform hypertree decomposition and measure the hypertree width $hw(H)$. This part takes **80%** of project full marks.

As we show at the beginning of Section 1.3, $tw(\mathcal{H})$ may vary for $\mathcal{Q}(H)$ representing a class of queries with their hypergraphs the same as $U(c) \bowtie N(c, a, d) \bowtie E(c, a)$. For example, acyclic queries $U(c) \bowtie N(c, a, d, e) \bowtie E(c, a)$ and $U(c) \bowtie N(c, a, d, e, f) \bowtie E(c, a)$ both belong to $\mathcal{Q}(H)$ but their treewidth is 3 and 4, respectively. To ensure queries with the same property, e.g., acyclic, have the same width [5], we exploit the intuition shown in Section 1.3.1 by measuring the number of hyperedges "cover" each bag in the tree decomposition instead of the bag size itself (leads to treewidth).

Definition 8 (Hypertree [GGS14])**.** *A hypertree for a hypergraph H is a triple $\langle T, \chi, \lambda \rangle$, where $T = (V(T), E(T))$ is a rooted tree, and χ and λ are labeling functions that associate with each vertex $p \in V(T)$ two sets $\chi(p) \subseteq V(H)$ and $\lambda(p) \subseteq E(H)$. The width of a hypertree is the cardinality of its largest λ label, i.e., $\max_{p \in V(T)} |\lambda(p)|$.*

Note that unlike width in Definition 6, we change the definition of width here: instead of measuring $|\chi(p)|$, we now use $|\lambda(p)|$ as the width. Thus, (generalized) hypertree decompositions are similar to tree decompositions, but for the associated notion of width, which is determined by a minimum hyperedge covering of the sets of nodes in the χ labeling.

Definition 9 (Generalized Hypertree Decomposition [FGLP21])**.** *A generalized hypertree decomposition (GHD) of a hypergraph $H = (V(H), E(H))$ is a tuple $\langle T, (B_u)_{u \in V(T)}, (\lambda_u)_{u \in V(T)} \rangle$, such that $T = \langle V(T), E(T) \rangle$ is a rooted tree and the following conditions hold:*

1. *$\forall e \in E(H)$: there exists a node $u \in V(T)$ with $e \subseteq B_u$;*
2. *$\forall v \in V(H)$: the set $\{u \in V(T) \mid v \in B_u\}$ is connected in T (Thus, T is a tree decomposition of H); and*
3. *$\forall u \in V(T)$: λ_u is defined as $\lambda_u : E(H) \to \{0, 1\}$ with $B_u \subseteq B(\lambda_u)$ where $B(\lambda_u)$ is a set that contains all vertices covered by λ_u, i.e., $B(\lambda_u) = \{v \in V(H) \mid \lambda(e) = 1$ for $e \in E(H), v \in e\}$.*

Compared to Definition 5, the first two criteria are completely the same. The new addition, the last criterion conveys the intuition we discussed earlier: using a set of hyperedges to cover vertices (i.e., attributes) shown in each bag B_u for $u \in V(T)$. Compared to Definition 8, despite the definition of λ looks different, they are the same: we can view λ from Definition 9 as a set $\lambda \subseteq E(H)$ (namely, the set of edges e with $\lambda(e) = 1$), which is the same as λ in Definition 8. Thus, the result of GHD is a hypertree with constraints

[5] For additional motivation of using (generalized) hypertree decomposition over tree decomposition, see [GGLS16].

listed in Definition 9 [6]. If you're familar with *edge cover*, λ from Definition 8 follows the edge cover definition precisely and λ from Definition 9 is a *linear programming (LP)* formulation of the same concept.

Aside note, for any $v \in V(H)$, if $v \in B(\lambda_u)$ but $v \notin B_u$, then v is "ineffective" for u and do not count with respect to the connectedness condition [GGS14, GGLS16]. For example, $B_1 = \{a, b, c\}$ and $\lambda_1 = \{R(a, b), S(b, c, d)\}$, then d is "ineffective", which can be projected out: $\lambda_1 = \{R(a, b), \pi_{b,c}S\}$. Thus, when we check connectedness property for d, we should not count node associated with B_1. This can be seen from Definition 9 but we want to emphasize this point.

From practical perspective, a node in GHD captures which attributes should be retained (projection with B_u) and which relations should be joined (with λ_u) [ALT+17].

Definition 10 (Generalized Hypertree Width [FGLP21, GGS14]). *The width of a GHD is* $\max_{u \in V(T)} |\sum_{e \in E(H)} \lambda_u(e)|$, *i.e., the maximum number of hyperedges associated with a node* $u \in V(T)$ *among all nodes in* T. *The generalized hypertree width of* H, *denoted by* $ghw(H)$, *is the minimum width over all GHDs of* H.

As noted in [GGLS16], a problem with GHD is that for fixed constants $w \geq 3$, it is NP-hard to decide whether for a query Q, $ghw(H_Q) = w$. In other words, a query with width w cannot be recognized in polynomial time. This is not ideal according to [GGLS16]. Hypertree decomposition is to fix this problem with one more requirement on GHD.

Following notation from [FGLP21], for a node u in T, we write T_u to denote the subtree of T rooted at u. By slight abuse of notation, we write $u' \in T_u$ to denote that u' is a node in the subtree T_u of T. We define $V(T_u) \coloneqq \cup_{u' \in T_u} B'_u$, the set of all vertices (attributes, variables) appeared in the bags of the subtree of T rooted at u.

Definition 11 (Hypertree Decomposition [FGLP21]). *A hypertree decomposition (HD) of a hypergraph* $H = (V(H), E(H))$ *is a GHD, which in addition also satisfies the following condition, called descendant condition or special condition [GGLS16]:*

1. $\forall u \in V(T) : V(T_u) \cap B(\lambda_u) \subseteq B_u$

Descendant condition suggests that if we have to deal with some attribute a in T_u, then we must immediately care about it in u if possible, i.e., if a appears in some hyperedge in λ_u.

Definition 12 (Hypertree Width [FGLP21]). *Since HD is a special case of GHD, hypertree width of* H, *denoted by* $hw(H)$, *follows the same definition as* $ghw(H)$.

Figure 8 shows a GHD of $R(a, b) \bowtie S(b, c) \bowtie T(c, a)$ (compared to tree decomposition in Figure 6 to see the difference.) In fact, this is a HD because $V(T_1) = \{a, b, c\}$, $B(\lambda_1) = \{a, b\}$, and $V(T_1) \cap B(\lambda_1) = \{a, b\} \subseteq B_1 = \{a, b\}$. Similarly, $V(T_2) = \{a, b, c\}$, $B(\lambda_2) = \{a, b, c\}$, and $V(T_2) \cap B(\lambda_2) = \{a, b, c\} \subseteq B_2 = \{a, b, c\}$. We can do the same checking for B_3.

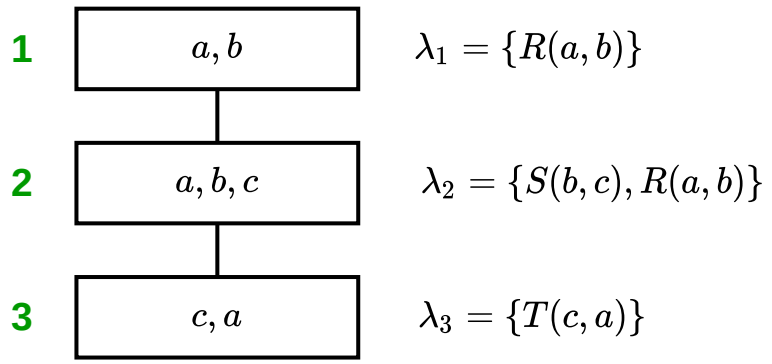

Figure 8: A GHD of $R(a, b) \bowtie S(b, c) \bowtie T(c, a)$.

[6] For awareness, hypertree definition is slightly different depending on the literature [Dec03, GGS14, GGLS16, FGLP21]. Difference between [GGS14] and [GGLS16] is on whether including constraints listed in Definition 9 as part of hypertree definition. [FGLP21] workarounds this difference by not emphasizing the concept of hypertree in GHD definition; as we have shown, GHD definition does not depend on definition of hypertree. Dechter's definition is on H instead of $\langle T, \chi, \lambda \rangle$ [Dec03].

Additional examples can be seen at [GGS14] Example 3.2 and [GGLS16] therein.

Instruction. The steps to finish this part are exactly the same as what described in Section 2. The only difference is that, instead of implementing and running tree decomposition algorithm, you need to implement hypertree decomposition algorithm. The algorithm is described in [GS09]. However, please feel free to reference [FGLP21, GGLS16, GGS14] on the comments about hypertree decomposition computation. In particular, [FGLP21] open sourced the implementation of [GS09] with a few enhancements:

https://github.com/TUfischl/newdetkdecomp

Thus, your task can be simplified to adapting existing implementation to work with the parsing result from the `SQL` parser of your choice.

2 Project Steps

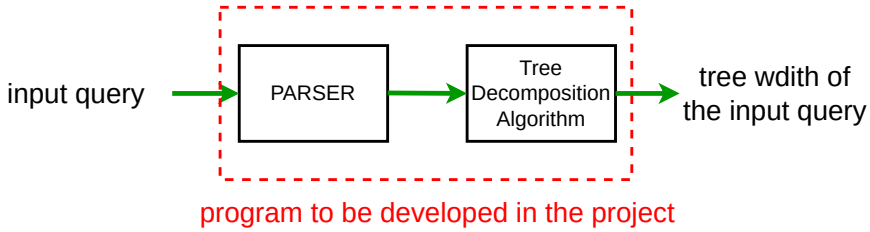

Figure 9: Program architecture to be developed for the project.

Figure 9 shows the program that is to be developed as part of the project. Let us walk through the steps to finish this project in details.

1. Parser
   (a) **Relational.** We *require* you to use the one of the following two parsers for the relational project:
      - https://github.com/JSQLParser/JSqlParser
      - https://github.com/hyrise/sql-parser

      If you find this is difficult to work with for technical reason, please let us know immediately with written reason before switching to another one.
   (b) **Graph.** You have two options for the project both are based on Cypher, the query language developed by Neo4j.
      i. Use the Cypher parsing module:
         https://github.com/opencypher/front-end
      ii. Neo4j source code:
         https://github.com/neo4j/neo4j

      Essentially, both options require you to use the Neo4j parser. However, we offer the flexibility in case you find one source code is easier to set up than the other. For Neo4j source code, you are allowed to use older version (i.e., commits) if compiling latest version (i.e., passing all the unit tests with the source code) is difficult.
2. Benchmarks. We require you to measure queries from the following benchmarks. Unless noted, all queries from the benchmarks listed need to be measured.
   (a) **Relational.** LSQB [MLK+21], TPC-H [(TP] (3 queries with the largest number of joins), JOB [LRG+18]
   (b) **Graph.** LSQB [MLK+21], One benchmark picked from shorturl.at/ekoFS

Benchmarks	Queries	$tw(H)$	(Extra Credits) $tw(H')$	(Extra Credits) $hw(H)$
TPC-H				
...				

Table 1: An example table of results to be reported

3. Implement an algorithm for finding treewidth. This includes generate the expected input for the algorithm from the parsed AST of queries. This part you have the freedom to explore. Potential algorithms to find a tree width k. shorturl.at/ioKLT lists quite a few but I would recommend to start with following four. In particular, start with either [SG97] or [KBJ19] ([KBJ19] has existing source code available), then [ACP87], and lastly [Bod93].

   One way to verify your implementation is that you can run it against some examples and compare with your manual calculation.

4. Feed queries into parser and then the parsed AST to the algorithm to produce treewidth automatically. At this step, we put everything together. Details to think about is

   (a) How to feed queries to parser? Use CLI or directly code into a program.

   (b) How to feed output of parser to the algorithm you choose.

   (c) How to report algorithm output and produce treewidth result?

   The answers to these questions will be answered as part of the report.

3 Deliverable

The deliverable contains two parts: a project report (typeset in LaTeX) and aforementioned program that you will develop. We will detail in the following.

1. The project report will need to contain the following:

   (a) A table of results. A possible template is shown in Table 1.

   (b) To promote reproducible research, you need to document the following as part of report

      i. Environment that you run the benchmarks (OS, hardware information)
      ii. Your setup of benchmarks (detailed steps)
      iii. Is there any challenges to use any of the benchmarks (if so, how you overcome it)?
      iv. Is there any modification to the benchmarks (if yes, what are they)?
      v. Highlight key implementation details to your program (answer to the questions in Item 4 above)
      vi. What source code you use with what commits (list commit ids)? How you setup the source code?
      vii. What tree decomposition algorithm you implement? Describe the algorithm and your implementation. What's the challenge to implement the algorithm. Please provide additional references you use.
      viii. How to use your program to obtain the benchmark results?
      ix. What's your observations based on the benchmark results? Anything that surprises you?

2. Program you developed with detailed instructions on reproducing the results reported in your write-up. The steps will be needed to include as a `README` file. Your program should be executed in a Linux/Unix environment. Please feel free to reuse your writings in your report. The central goal is that anyone can reproduce your result using your instructions easily. **Your grade is tie to the reproduction**

of your result. We will run your program according to the instructions provided and reproduce your result and check if it is aligned with what is being reported. All source code you write is subject to examination. Adoption of good software engineering practice is at the core of the examination.

References

[ACP87] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of Finding Embeddings in a k-Tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

[AGG07] Isolde Adler, Georg Gottlob, and Martin Grohe. Hypertree width and related hypergraph invariants. *European Journal of Combinatorics*, 28(8):2167–2181, 2007. EuroComb '05 - Combinatorics, Graph Theory and Applications.

[ALT+17] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4), October 2017.

[BFMY83] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the Desirability of Acyclic Database Schemes. *Journal of the ACM (JACM)*, 30(3):479–513, 1983.

[Bod93] Hans L. Bodlaender. A Linear Time Algorithm for Finding Tree-Decompositions of Small Treewidth. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, page 226–234, New York, NY, USA, 1993. Association for Computing Machinery.

[BRV16] Pablo Barceló, Miguel Romero, and Moshe Y. Vardi. Semantic Acyclicity on Graph Databases. *SIAM Journal on Computing*, 45(4):1339–1376, 2016.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, page 77–90, New York, NY, USA, 1977. Association for Computing Machinery.

[Dec03] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, USA, 2003.

[FGLP21] Wolfgang Fischl, Georg Gottlob, Davide Mario Longo, and Reinhard Pichler. HyperBench: A Benchmark and Tool for Hypergraphs and Empirical Findings. *ACM J. Exp. Algorithmics*, 26, jul 2021.

[GGLS16] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions: Questions and Answers. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '16, page 57–74, New York, NY, USA, 2016. Association for Computing Machinery.

[GGS14] Georg Gottlob, Gianluigi Greco, and Francesco Scarcello. *Tractability: Practical Approaches to Hard Problems*, chapter Treewidth and Hypertree Width, pages 3–34. Cambridge University Press, 2014.

[GM14] Martin Grohe and Dániel Marx. Constraint Solving via Fractional Edge Covers. *ACM Transactions on Algorithms (TALG)*, 11(1):1–20, 2014.

[GMUW08] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, USA, 2 edition, 2008.

[GS09] Georg Gottlob and Marko Samer. A Backtracking-Based Algorithm for Hypertree Decomposition. *ACM J. Exp. Algorithmics*, 13, feb 2009.

[GSS01] Martin Grohe, Thomas Schwentick, and Luc Segoufin. When is the Evaluation of Conjunctive Queries Tractable? In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*, STOC '01, page 657–666, New York, NY, USA, 2001. Association for Computing Machinery.

[KBJ19] Tuukka Korhonen, Jeremias Berg, and Matti Järvisalo. Solving Graph Problems via Potential Maximal Cliques: An Experimental Evaluation of the Bouchitté–Todinca Algorithm. *ACM J. Exp. Algorithmics*, 24, feb 2019.

[KNRR16] Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Joins via Geometric Resolutions: Worst Case and Beyond. *ACM Trans. Database Syst.*, 41(4), November 2016.

[Kol16] Phokion G. Kolaitis. Logic and Databases, August 2016.

[Kou22] Paris Koutris. Lecture 5: Query Decompositions. Online, February 2022.

[LRG+18] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. Query Optimization through the Looking Glass, and What We Found Running the Join Order Benchmark. *The VLDB Journal*, 27(5):643–668, October 2018.

[Mar13] Dániel Marx. Tractable Hypergraph Properties for Constraint Satisfaction and Conjunctive Queries. *J. ACM*, 60(6), November 2013.

[MLK+21] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. LSQB: A Large-Scale Subgraph Query Benchmark. In *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences and Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA '21, New York, NY, USA, 2021. Association for Computing Machinery.

[MW95] Alberto O. Mendelzon and Peter T. Wood. Finding Regular Simple Paths in Graph Databases. *SIAM Journal on Computing*, 24(6):1235–1258, 1995.

[SG97] Kirill Shoikhet and Dan Geiger. A practical algorithm for finding optimal triangulations. In *AAAI-97 Proceedings*, 1997.

[(TP] Transaction Processing Performance Council (TPC). TPC-H Benchmark. Online. Accessed on 11-18-2021.

[Yan81] Mihalis Yannakakis. Algorithms for Acyclic Database Schemes. In *VLDB*, volume 81, pages 82–94, 1981.