Fundamentals, Significance and Practicality of Treewidth and Tree Decompositions

Taylor Cox
COMP 4060: Graph Theory
University of Manitoba
Winnipeg, Manitoba
Email: coxt3@myumanitoba.ca

***Abstract*—There is a large set of NP-Hard problems for which there are polynomial time solutions on graphs of bounded treewidth. In 1992, Bodlaender gave a linear time algorithm to determine whether a graph has treewidth of a fixed constant k or less. Despite the fact that the treewidth problem is fixed-parameter tractable, the implementation of a general or fixed-k treewidth algorithm has been largely unsuccessful in the graph theory literature. The absence of a comprehensive, practical algorithm for determining the treewidth of a graph leads to the conclusion that fixed-parameter tractibility may not be a sufficiently descriptive characterization of the treewidth problem and related problems.**

***Index Terms*—Treewidth, Tree Decomposition, Fixed Parameter Tractibility**

I. Introduction

A key result of Robertson and Seymour's Graph Minor Theory is the notion of treewidth [1]. Let $G = (V, E)$ be an arbitrary graph. Informally, the treewidth of G $tw(G)$ is an indicator of how similar G is to a tree. Formally, $tw(G)$ is the minimum width across all possible tree decompositions of G. A tree decomposition is an organization of V into a collection of bags, where each bag contains one or more vertices and the union of all bags is equal to V. The bags are connected such that they form a tree. The width of a tree decomposition is the cardinality of its largest bag minus one. Treewidth-related definitions are further explored in section V.

Many NP-Complete and NP-Hard graph problems are solvable in polynomial time for graphs of bounded treewidth [2]. Such problems include Three-Colorability, Max-Clique and Minor Containment. The substantial theoretical significance of the treewidth property has motivated extensive research into approaches for determining the treewidth of a graph. For a general graph G of nonzero genus, it is NP-Complete to determine whether $tw(G) \leq k$ for some k. When limited to the planar graphs, is not known whether the treewidth problem is in P or NP [3]. Approximation, Heuristic, Dynamic Programming and Fixed-Parameter approaches have all been developed for the purpose of finding the treewidth for general graphs, with varying degrees of success [4].

Two treewidth algorithms in terms of their programmatic and computational complexity. The first algorithm is Bodlaender's 1992 fixed-paramater treewidth algorithm [5], and is ultimately determined to be impractical for usage outside of theoretical computer science. The second algorithm explored is Bodlaender's 2012 dynamic programming algorithm [6]. A C implementation of this algorithm is benchmarked against the implementation provided in Sage, a standard mathematical software package [7]. Comparative results indicate a need to employ further optimizations in order to decrease the algorithm's runtime. However, the potential for improvement to the algorithm is limited due to its exponential nature. The Sage benchmarks themselves do not run in graphs of more than 100 vertices.

The remainder of this report is structured as follows: Section II captures a formal description of the treewidth problem itself. Sections III and IV describe the background and related literature to the treewidth problem, and outline the motivation for implementing algorithms for treewidth. In sections V through VII, implementations of Bodlaender's 1992 and 2012 algorithms will be described and critically evaluated. Finally, section VIII will deliver the key conclusions of this project, accompanied by possible directions of future work from theoretical and implementation perspectives.

II. Problem Description

The treewidth problem is as follows: "Given a graph G and a positive, nonzero integer k, does G have treewidth at most k?". The problem is known to be NP-Complete. The treewidth of a graph $tw(G)$ is the minimum width across all possible tree decompositions of G. A *tree decomposition* of a graph is as follows. Let $G = (V, E)$ be an arbitrary graph of more than one vertex. Then the tree decomposition $D = (X, T)$ is a 2-tuple containing X and T. X is a collection $X_1, X_2, ..., X_m$ of cardinality m. Each subset X_i contains some subset of V. T is a tree that describes the connections between each subset. The subsets are often referred to as *bags*, and obey the following properties with respect to G and T:

1) $\cup_{X_i} = V$
2) Let X_i and X_j be two distinct members of X. If there exists some vertex u such that $u \in X_i$ and $u \in X_j$, then there exists a path in T from X_i to X_j, such that each X_k on the path also includes u.
3) Let u and v be two vertices in G. If there exists an edge uv, then there exists some bag X_i that contains both u and v.

The *width* of a tree decomposition $w(D)$. is the cardinality of its largest bag, minus one. The treewidth property is therefore always strictly less than the number of vertices in the graph. Subtracting one results in $tw(G) = 1$ for trees, which is a more desirable outcome than the treewidth of a tree being two. It is evident from the definition prodivded that multiple tree decompositions are possible for most graphs. However, some graphs induce only one tree decomposition. Since the treewidth of a graph is its minimum possible tree decomposition width, there exist some graphs that have only one possible treewidth.

Theorem 1. $tw(K_n) = n - 1$

Proof: Induction is used on n.

Base case: $n = 3$. Let $G = K_n$ and $D = (X, T)$ be the tree decomposition of G. Assume $tw(G) < n - 1$. Then each bag in X covers exactly two vertices (there are two such bags and they must be connected). Therefore, there exists an edge uv such that neither bag in X contains both u and v. Let Y be a new bag consisting of u, v. Since uv is the only edge not present in the decomposition, Y will be connected in T to both existing bags in X, one of which containing u and the other containing v. However, the two existing bags in the decomposition are already connected. Therefore, connecting Y to both bags induces a cycle in T. Contradiction. uv must be introduced to the decomposition by appending v into the bag containing u. The result is a bag containing n vertices. Since a bag in X contains n vertices, no other bags are needed. The treewidth of K_n for $n = 3$ is therefore 2.

Induction case: Assume $tw(K_{n-1}) = n - 1$. Let $D = (X, T)$ be the tree decomposition of K_{n-1}. K_n is generated from K_{n-1} by introducing a vertex u and connecting it to all vertices. Partition the endpoints of edges in K_n containing u into two or more bags Each new bag will also contain u, and is connected to the original bag in D. Since each bag contains u, the bags are also connected to eachother in a path. This induces a cycle in T. Contradiction. The only way to introduce u is by composing a single bag containing the endpoints of all edges departing from u. The result is a single bag containing all n vertices, which must have treewidth $n - 1$. □

A tree decomposition is considered minimal when none of its bags may be reduced in size without affecting the decomposition's correctness. Since $tw(G)$ is the minimum decomposition width across all possible tree decompositions of G, $tw(G)$ is equal to the width of the *minimal* tree decomposition of G. Figure (1) shows the tree decomposition of a graph on 7 vertices, accompanied by a minimal tree decomposition. The treewidth of the graph shown is 3. Let D_{min} be the minimal decomposition of G. Since $tw(G) = w(D_{min})$, knowing the treewidth of G is equivalent to knowing its minimal tree decomposition. Since non-minimal tree decompositions are equally or less powerful than minimal tree decompositions, only minimal decompositions will be considered for the remainder of this report.

Theorem 2. *Let C be the max clique in G. Then $tw(G) \geq |C| - 1$.*

Proof: Let $D = (X, T)$ be the (minimal) tree decomposition of G. By (1) and (2) of the tree decomposition definition, the endpoints of each edge in C are included in some X_i, and the bags containing each vertex in C are connected. Since an edge exists between every vertex pair in C, the bags formed by C must be interconnected. Interconnected bags form a cycle. Contradiction. The vertices of C are part of or form a single bag Y. If Y is the largest bag and $Y = C$, $tw(G) = |C| - 1$. Otherwise, $tw(G) > |C| - 1$. □

The treewidth problem is NP-Complete, and the minimal tree decomposition problem is NP-Hard. Even to the trained eye, instances of either problem involving more than a small number of vertices generally require computer aid. Since treewidth is derived from tree decompositions, the treewidth problem may only be solved by finding a minimal tree decomposition. The continued research efforts in the treewidth problem are motivated by the substantial benefits of the treewidth property. The background and motivations behind solving the treewidth problem are now discussed.

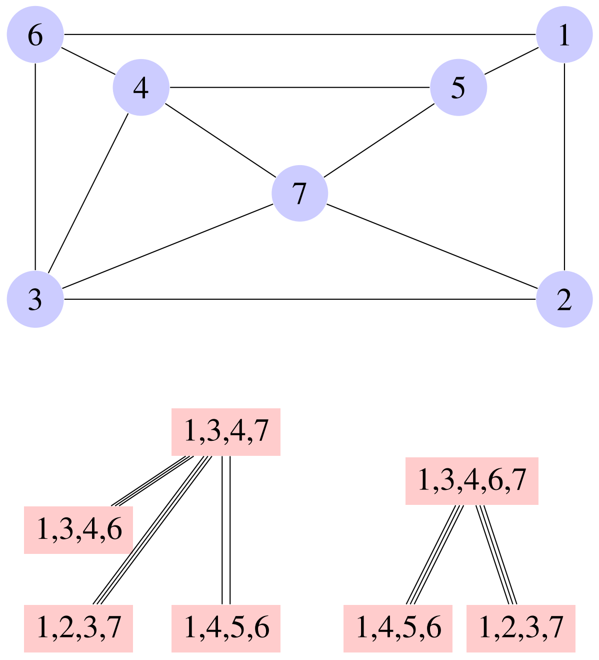

Fig. 1. Two tree decompositions of a graph. One is minimal and the other is not. The graph has treewidth 3.

III. Background

Treewidth and tree decompositions have been thoroughly explored in the literature. Research perspectives include range from the purely theoretical to the purely practical points of view. The first recorded examination of the treewidth property is from Bertelé et al in 1972 [8]. Treewidth was originally referred to as graph *dimension*, and was known to be a valuable property with respect to the solvability of NP-hard problems. The treewidth literature is primarily categorized into works developing methods for determining treewidth, and works exploring the power of the treewidth property. The remainder of this section is divided the same way.

A. Determining Treewidth

After initial investigation by Bertel, the treewidth property remained unexplored for some time. In 1984, Robertson and Seymour gave a re-examination of treewidth as part of Graph Minor Theory [1]. While Robertson and Seymour had shown the theoretical and algorithmic significane of treewidth, no algorithms yet existed to determine $tw(G)$. By 1987, Arnborg had proven that the treewidth problem is NP-Complete [9]. However, Bodlaender gave an algorithm in 1992 to determine a fixed-parameter variant of the treewidth problem [5]. Bodlaender's algorithm answered the following question: "given a graph G and a *fixed constant* k, does G have treewidth k or less?". The algorithm proved that the treewidth problem is fixed-parameter tractable. Bodlaender was able to prove that when k is held fixed (and thus is not captured in the O(*) measurement of the algorithm), the algorithm runs in O(n) against the size of the graph. However, the algorithm is known to have a recursive depth of up to k^8 and depends on a constant as large as k^{k^3} [10]. Therefore, Bodlaender's algorithm is generally considered impractical for implementation [11]. Reed gave an improvement to Bodlaender's algorithm with a recursion depth of up to k^2 [14]. However, Reed's algorithm also bears a prohibitive dependency on large hidden constants [15].

Other alternatives in the literature have been explored for determining $tw(G)$. These alternatives focus on heuristics for treewidth bounding. Many modern heuristics for treewidth rely on the notion of a *perfect elimination ordering*. For $G = (V, E)$, A perfect elimination ordering is an ordering π of V where the ith vertex of π is simplical in the subgraph formed by all vertices from i to n. A vertex is simplical if its neighbours induce a clique. Algorithms for generating perfect elimination orderings also serve as heuristics for treewidth upper bounding.

One heuristic based on simplical vertices and eliminiation orderings is the minimum degree heuristic [12]. In this heuristic, choose a vertex v of minimum degree, introduce edges such that v is simplical. Let the new graph be G'. Build a tree decomposition in G' containing some bag Y that covers each neighbour of v (by theorem 2, every clique in G forms a bag in X). A new bag - one that contains v and its neighbours - is attached to Y. This gives a tree decomposition in G. This heuristic may not produce a minimal tree decomposition, but it will give an upper bound for treewidth.

The algorithmic focus of this report is on exact algorithms for treewidth. The standard algorithm for determining treewidth is developed by Bodlaender, in 2012 [6]. The algorithm given uses dynamic programming, and functions similar to the Held-Karp Travelling Salesman Problem dynamic programming algorithm. The algorithm begins with an empty treewidth of infinite weight, and tightens the bound on the treewidth until a minimal decomposition is found. The algorithm takes advantage of the minimax problem that arises in treewidth computation. The algorithm uses that fact treewidth is the *maximum* cardinality of the *minimum* tree decomposition to develop a principle of optimality as needed for a dynamic programming algorithm. The algorithm relies on the max clique of G as a stopping condition, since the $tw(G)$ cannot be exceeded by the cardinality of the max clique. Bodlaender's 2012 dynamic programming algorithm produces exact values for treewidth, but runs in O(2^n) time.

Many algorithms exist to determine or approximate the treewidth of the graph. Since the treewidth problem is NP-Complete and the tree decomposition is NP-Hard, much work has been accomplished to find appropriate limitations to the treewidth problem. These problems include fixed parameter algorithms, heuristics, and bounding approaches. The key algorithms for treewidth are Bodlaender's 1992 and 2012 algorithms. Bodlaender's 1992 algorithm gives a general approach for solving the treewidth problem, but is not practical for implementation. In contrast, Bodlaender's 2012 algorithm is appropriate for implementation, but carries and exceptionally slow runtime. Section V covers both of Bodlaender's algorithms in further detail.

B. Treewidth Significance

Robertson and Seymour use treewidth as part their proof of Wagner's Conjecture. Wagner's conjecture (also known as the Robertson-Seymour theorem) states that every minor-closed family of graphs is characterizable by a finite set of forbidden minors, also referred to as obstructions. Combined with the proof (also by Robertson and Seymour) that there exists a polynomial time algorithm for the fixed minor containment problem, it is clear that all minor-closed graph properties are polynomial time membership testable. Without the notion of treewidth, these results would not be obtainable. Robertson and Seymour first determined that planar graphs of bounded treewidth formed a well-quasi-ordering: for any planar obstruction, the family of planar graphs characterized by that obstruction have a treewidth bounded by the size of the obstruction [13]. Robertson and Seymour then generalized the bounded treewidth result to form well-quasi-orderings for families of general graphs closed under taking minors. Bounded treewidth leads to polynomial time algorithms for fixed minor containment, graph embeddability, and other algorithms concerning minor-closed properties. The treewidth property itself is also minor-closed. Therefore, taking the minor of a graph with treewidth at most k gives another graph of treewidth at most k.

Outside of graph minor theory, the treewidth parameter is known to give polynomial time solutions for many NP-Hard problems. The algorithmic potential treewidth was recognized by both Bertelé and Robertson and Seymour. However, no progress was made in the algorithmic benefits of treewidth until Bodlaender explored the usage of bounded treewidth for dynamic programming [17]. It was determined that graphs which are sufficiently "tree-like" (those of bounded treewidth) lend themselves to be processed efficiently with dynamic programming algorithms. Dynamic programming soltuions for graphs of bounded treewidth are known for problems including Maximum independent set and Hamiltonian Cycle. Similar results are produced by Arnborg, who showed that partial k-trees (graphs of treewidth bounded by a constant k) have *linear time* solutions for many NP-Hard problems. These solutions are also based on dynamic programming [16]. The list of solvable NP-Hard problems for graphs of bounded treewidth is highly expansive. To illustrate the power of the treewidth property, polynomial time solvability proofs for bounded-treewidth variants of 3-colorability and max clique are now given. Note that the resulting algorithms are linear time with respect to n, with a constant value dependent on $tw(G)$.

Theorem 3. *Let $G = (V, E)$ be a graph with a known tree decomposition $D = (X, T)$ of width w. 3-colorability can be solved on G in $3^w * O(n)$ time.*

Proof: Dynamic Programming is used on X. Choose a root bag X_1 arbitrarily. Partition the vertices of X_1 into 3 color classes based on their connectivity. Extend the coloring to connected bags from X_1 in T. Repeat for each bag, backtracking as needed in case of conflict. Either an unresolvable conflict is found, or a coloring of the vertices is given. $\square$

Theorem 4. *Let $G = (V, E)$ be a graph with a known tree decomposition $D = (X, T)$ of width w. Max-clique can be solved on G in $2^w * O(n)$ time.*

Proof: For each bag in X, compute the maximum clique in X using a standard backtracking algorithm. Sufficient algorithms include [18]. The maximum clique across all bags is the maximum clique of G. Since w is a known constant parameter, execution of the maximum clique algorithm is excluded from the $O(*)$ complexity. $\square$

The treewidth property is highly significant for the development of solutions to NP-Hard problems. The list of solvable NP-hard problems for graphs of bounded treewidth is highly extensive. As proved by Courcelle in 1990 [19], *all* graph properties describable in extended monadic second order logic are linear-time decidable for graphs of bounded treewidth. Second order logic is the logical system that permits quantification over sets and variables (first order logic exclusively permits the quantification over individual variables). Monadic second order logic disallows the quantification over subsets. A second order logic system is *extended* when applied to the logic of graphs. Extended monadic second order (EMSO) logic permits the quantification over vertices, edges, vertex-sets, and edge-sets.

Theorem 5 (Courcelle's Theorem)**.** *Let $G = (V, E)$ be a graph with a known tree decomposition $D(X, T)$ of width w. Let M be some extended monadic second order logic-describable property formulated as a quantificational sentence. M is linear time decidable on G.*

proof: The proof by Courcelle relies on tree automata. An algorithm is given that accepts G and M and decides whether G satisfies M. M can be linear-time translated into a sentence M' such that D satisfies M' iff G satisfies M. The proof is given in detail in [19]. $\square$

Many of the algorithms that are solvable for graphs of bounded treewidth rely on deriving solutions for subproblems in each bag. Problems independently solvable on bags of a tree decomposition are therefore parallelizable. In the maximum clique case, up to n processes are able to independently compute the maximum clique of each bag. Processes need only communicate to determine the globally maximum clique. It must also be noted that the linear time algorithms proposed rely on exponential contants with respect to w. Exponential dependence on w is not unique to the example problems shown. The exponential constant factors in bounded treewidth "linear time" algorithms results in the impracticality of such algorithms for large values of w.

Despite the impracticality of some tree decomposition-based algorithms, the theoretical benefits of the treewidth property are more than sufficient to warrant research efforts into the development of algorithms for determining treewidth and generating tree decompositions. Since every EMSO-describable property is linear time decidable for graphs of bounded treewidth, the exploration of treewidth and tree decomposition algorithms is a highly rewarding endeavour.

IV. Motivation

The theoretical benefits of the trewidth property are extensive. However, the generation of treewidth and tree decompositions for arbitrary graphs is a monumental task. Despite the fact that the treewidth problem is known to be fixed-parameter tractable, there are no approaches for determining treewidth in the literature with the property of being practical for implementation and flexible for arbitrary graph size or structure. The motivation of this work is to (1) highlight the discrepency between the theoeretical benefits and practical approachability of treewidth, and (2) develop efforts towards the reconciliation of this discrepency. The theoretical applications of treewidth are considerably less meaningful without sufficient means to determine treewidth to begin with.

V. Solution Techniques

The separation between the theoretical significance and practical usability of treewidth is ideally reconciled with the development of efficient algorithms and heuristics for determining treewidth and tree decompositions. The theoretical benefits of the treewidth property arise in graphs of upper-bounded treewidth. Therefore, the primary problem of interest is treewidth upper bounding. Lower bounds on $tw(G)$ are explored in [20] and [21].As described in section II-A, a variety of approaches for determining treewidth have been explored in the literature. The most common algorithm for treewidth is the 1992 linear time fixed-parameter algorithm proposed by Bodlaender [5]. A general exponential-time treewidth algorithm also exists and is also first discovered by Bodlaneder [6].

A. The fixed-parameter treewidth algorithm

Bodlaender's linear-time algorithm relies on the fact that treewidth is minor-closed. Let G be a graph and H its minor. If $tw(G) \leq k$, then $tw(H) \leq k$. Therefore, a graph of arbitrary size may be reduced to a much smaller graph on which a finite or brute-force algorithm may easily be used. The only requirement is that the graph must be shrunk in a manner that does not alter the tree decomposition.

The algorithm is divided into two cases: for an input graph G, G either have sufficiently many or only few friendly vertices. The specific definitions of "sufficiently many" and "low degree" are dependent on constant parameters with respect to k, and are explored more thoroughly in Bodlaender's original publication. A vertex is deemed friendly if it is of low degree and all of its neighbours are also of low degree. If sufficiently many friendly vertices are found, a maximal matching M of G is computed and all edges in M are contracted. The algorithm then executes recursively against the contracted graph G'. Otherwise, only a small number of vertices are found in G. In this case, an edge is added between every vertex pair with $k+1$ common low degree neighbours. The graph with edges added is G_i. The simplical vertices in G_i are removed, creating a graph G'. The algorithm then executes recursively on G'. The two cases of the algorithm are now examined in closer detail.

1) Maximal Matching Contraction: The first case of the linear time treewidth algorithm occurs when there are sufficiently many friendly vertices. In this case, a new graph is generated based on contracting the edges of a maximal matching in G. The matching-contracted graph is then sent to a recursive call of the algorithm. The following is an implementation of how a maximal matching may be computed and contracted for some graph.

Algorithm 1 Generate a Maximal Matching of G

```
 1: procedure MaximalMatching($G$)
 2:     $M$ ←LinkedList
 3:     visited ← [0]*$|V|$
 4:     for $u \in V$ do
 5:         if $u$ not visited then
 6:             visited[$u$] = 1
 7:             for $v \to u$ do
 8:                 if $v$ not visited then
 9:                     visited[$v$] = 1
10:                     add edge $uv$ to $M$
11:                     break
12:                 end if
13:             end for
14:         end if
15:     end for
16:     return $M$
17: end procedure
```

The maximal matching of a graph is a matching that cannot be grown to accomidate additional edges. A greedy algorithm is therefore sufficient to compute the maximal matching of G. To contract an edge $uv \in G$, the edge uv and the vertex v are marked deleted. The adjacencies of G are stored as linked lists. Therefore, only one pointer reassignment is needed to append the adjacencies of u to the adjacancies of v. The adjacencies are then deduplicated in case u and v shared common neighbours. Determining M and contracting every edge $uv \in M$ can both be performed in linear time. Therefore, the first case of the linear time treewidth algorithm performs in O(n) in the context of a single recursive call.

Treewidth is a minor-closed property. Let G' be the graph generated by contracting the maximal matching. Since G' is a minor of G, the treewidth of G' is no more than the treewidth of G. However, the treewidth of G may be larger than the treewidth of G'. The tree decomposition given by G' must be altered into a tree decomposition of G'.

Theorem 6. *Let k and l be two integers. Let $G = (V, E)$ be a graph and $D = (X, T)$ be a tree decomposition on G of width l. If a tree decomposition $D' = (X', T')$ of G exists of width k, it can be found in linear time.*

proof: The proof is given by Bodlaender in [22]. The first case of the algorithm therefore produces tree decompositions of width bounded by k. □

In cases where G has a sufficient number of friendly vertices, a tree decomposition of treewidth k or less is producable in linear time. This covers the high number friendly vertices case within a single recursive call. Recursion continues until the graph is small enough to permit brute-force execution. If there is an insufficient number of friendly vertices, the simplical vertex removal case is applied.

2) Simplical Vertex Removal: This case of the algorithm occurs when there are insufficient friendly vertices. A new graph is generated by introducing edges between vertex pairs with $k + 1$ or more low degree neighbours in G. The resulting graph is denoted G_i, the *improved* graph. A new graph, G' is computed by removing every simplical vertex in G_i. These vertices are described as *i-simplical* as they are simplical in G_i but may not be simplical in G. The approach in this case is analagous to the minimum degree heuristic in [12]. Simplical vertices are a key aspect of tree decompositions as their neighbours will always be found in a single bag. The following pseudocode describes an approach to generating G_i and subsequently generating G' from G.

Algorithm 2 Generate G' from G

```
procedure ImproveGraph($G$)
    for $u \in V$ do
        for $v \in V - u$ do
            $L_u$ = LowDegreeAdj($u$)
            $L_v$ = LowDegreeAdj($v$)
            if $|L_u \cap L_v| \geq k + 1$ then
                Add edge $uv$ to $G$
            end if
        end for
    end for
end procedure

procedure RemoveSimplicals($G$)
    for $u \in V$ do
        if Clique(Adjacencies[$u$]) then
            Delete vertex $u$ from $G$
        end if
    end for
end procedure

procedure Clique($VertexSet$)
    clique ← []
    for $u \in VertexSet$ do
        for $v \in$ clique do
            if $u \not\to v$ then
                return False
            end if
        end for
        clique ← u
    end for
    return True
end procedure
```

The three procedures above together form the low-number-low-degree case of the algorithm. The improved graph is generated, and its simplical vertices are pruned. Simplical vertices are those whose neighbours induce a clique. Generating G_i from G can be completed in linear time. In its given implementation, generating G' from G_i does not. Let C be a set which is at first empty and let V be the vertices constituing a possible clique. If the current vertex in V has every memeber of C as an adjacency, v is introduced into C. Therefore, the total number of operations performed is $\sum_{i=1}^{|V|} i = O(n^2)$. However, a series of optimizations are proposed in the original publication that can reduce the run time to $O(n)$ [5].

Removing i-simplical vertices from G gives a graph with a tree decomposition upper-bounded by $k-1$. It must first be noted that if any i-simplical vertices have degree greater than k, the algorithm terminates as a clique of size $k+2$ or more will be produced, resulting in a tree decomposition of width greater than k. The recursive call to the algorithm gives a tree decomposition of G' with treewidth at most $k-1$. A decomposition D of G with width at most k is then generated.

Theorem 7. *Let $D' = (X', T')$ be the tree decomposition generated by G' with width at most $k-1$. D' is transformable into a tree decomosition of G in linear time with treewidth no greater than k.*

proof: D' is only missing the i-simplical vertices (as they were previously deleted). Since each i-simplical vertex has degree at most k, the treewidth of D' is strictly less than k. If the largest bag were a k-clique, the width would be $k-1$. For each i-simplical vertex v, create a new bag X_i containing v and its neighbours. Attach X_i to an $X_j \in X$ containing the neighbours of v (it is shown in the original publication that X_j exists [5]). Since the maximum treewidth before introducing X_i was $k-1$ and $|X_i| \leq k+1$, the new tree decomposition D has a treewidth upper bounded by k. $\square$

3) Analysis: The treewidth algorithm outlined is claimed to run in linear time with respect to the size of the graph. However, the data structure demands, recursion depth and constant-parameter depence make this runtime untenable. First, the algorithm demands specific optimizations in order to run in linear time. For example, the authors specify that a queue of stacks must be kept in order to ensure that i-simplical vertices can be detected in $O(n)$ time. At minimum, the data structure demands imposed by this work lead to a space complexity much higher than $O(n)$. At maximum, the data structure demands prevent the algorithm from truly running in linear time. In addition, the recursive depth is a major concern for the complexity of the algorithm. In their seminal book on parameterized complexity, Downey and Fellows note that the recursion in Bodlaender's 1992 algorithm is as deep as k^8. While the algorithm may run in linear time within a single recursive call, the enormous recursive depth of the algorithm quickly dismisses any notion that the algorithm runs in linear time. If concerns regarding recursion were to be aleviated, the algorithm still depends on a doubly exponential constant in terms of k. The constant parameter to the fixed-parameter algorithm is k^{k^3}. The algorithm quickly becomes impossible to execute for values of k greater than 2.

Despite its relatively simply nature, there are hidden costs to the standard treewidth algorithm that result in any implementation attempt (including the attempt undertaken as part of this work) to be prohibitively unfeasible. The algorithm has been noted in the literature as "not recommended for implementation" [23], and attempts to implement the algorithm have noted to be "unsuccessful" [24]. Therefore, further work must be undertaken to approach the treewidth problem from an algorithmic perspective.

The algorithm originally proposed by Bodlaender relies on a base case. When the graph is sufficiently small, a finite or brute force algorithm may be used. The most efficient algorithm to date is Bodlaender's 2012 dynamic programming algorithm, which runs in exponential time. It is noteworthy that the ideal base-case to Bodlaender's 1992 algorithm remained unpublished for two decades. Many heuristics for treewidth have since been developed, leaving this algorithm to remain as the standard finite approach. Bodlaender's 2012 algorithm is considered to be the standard algorithm for determining treewidth in the lowest possible exponential time.

B. The Exponential Time treewidth Algorithm

The exponential time algorithm proposed by Bodlaender in 2012 uses dynamic programming and runs similar to the Held-Karp algorithm for the travelling salesman problem [6]. Since treewidth is the maximum width of the minimum tree decomposition, a minimax optimization problem easily arises.

This optimization problem is used as the basis for a dynamic programming strategy. The algorithm begins with the trivial decomposition (all vertices in one bag) of width $n-1$. Each vertex v in the graph is then examined. Since every edge uv must appear in some bag X_i, the next bag to be created will have size equal to $v \cup S$, where S composes the vertices in the current bag (S begins as the empty set). If the new bag has size smaller than the current width, a new tree decomposition is created to account for the new bag. If the contents of the new bag were already present in a candidate tree decomposition, the width is decreased. Otherwise, the bag is inserted into the decomposition. The algorithm continues for each vertex until every $v \in V$ is accounted for.

1) Building Tree Decompositions: To grow the tree decomposition, the algorithm uses a depth first search to determine the number of vertices reachable by the candidate bag. Let v be the current vertex under inspection and let S be the current bag of the decomposition under inspection. The vertices that form the current bag are $v \cup S$. The number of reachable vertices to $v \cup S$ can be determined by a depth first search. If the number of reachable vertices is lower than the current maximal treewidth, the decomposition growing process continues. Otherwise, the next bag would have to have a larger width and is therefore discarded.

If the number of reachable vertices is lower than the current maximal treewidth, the decomposition growing process continues. Otherwise, the next bag would have to have a larger width and is therefore discarded.

2) Stopping Conditions: The algorithm uses properties related to max cliques for early-exit optimizations. The first of these optimizations is given in theorem 2. Let C be a max clique in G.

Pseudocode for the algorithm is given in its original publication [6]. A C implementation of the algorithm is completed as part of this work and is available at [25].

VI. Experimental Setup

VII. Experimental Results

VIII. Conclusions and Future Work

References

[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
[11]
[12]
[13]
[14]
[15]
[16]
[17]
[18]
[19]
[20]
[21]
[22]
[23]
[24]
[25]