Towards Practical Algorithms and Accurate Categorization for the Treewidth Problem

Taylor Cox
COMP 4060 Project Report
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. The treewidth problem is NP-Complete. However, there exists a linear time algorithm to determine whether a graph has treewidth upper-bounded by a fixed constant k. 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 tractability may not be a sufficiently descriptive characterization of the treewidth problem and related problems.**

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

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 subsets called 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. The fundamentals of tree decompositions are further explored in section V.

Many NP-Hard graph problems are solvable in polynomial time for graphs of bounded treewidth [2]. Such problems include Three-Colorability, Maximum Clique and Vertex Cover. 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 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 of general graphs, with varying degrees of success [4].

This work closely examines two treewidth algorithms in terms of their programmatic practicality and computational complexity. The first algorithm is Bodlaender's 1992 fixed-parameter 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 35 vertices. The dynamic programming approach also outperforms Sage on graphs of low symmetry.

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 relevant literature to the treewidth problem, and outline the motivation for implementing treewidth algorithms. 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 of m subsets of V $X_1, X_2, ..., X_m$ which may or may not be disjoint. T is a tree that describes the connections between the subsets. 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 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 $tw(G) = 1$ where G is a tree, which is a more desirable outcome than the treewidth of a tree being two. It is evident from the definition provided 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 decomposition width, there are some graphs that have only one possible treewidth. As a result, the treewidth of the complete graph K_n is known without requiring algorithmic validation.

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

Proof: Induction is used on n.

Base case: $n = 3$. If $n < 3$, G is a tree and therefore has treewidth 1. Let $G = K_3$ and $D = (X, T)$ be the tree decomposition of G. Since $tw(G) < n$, either $tw(G) = n - 1$ or $tw = (G) < n - 1$. If $tw(G) = n - 1$, the proof is complete. Assume $tw(G) < n - 1$ for $n = 3$. 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 $tw(K_{n-1}) = n - 2$. 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 each other in a path. This induces a cycle in T. Contradiction. The only way to introduce u is by constructing 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 of the graph. 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.

An additional known result related to treewidth concerns maximum cliques. The size of the maximum clique in G is a useful means to determine a lower bound on $tw(G)$.

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 encouraged by the substantial benefits of the treewidth property. The background to the treewidth problem and motivations for determining solutions are now discussed.

III. Background

Treewidth and tree decompositions have been thoroughly explored in the literature. Research perspectives 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

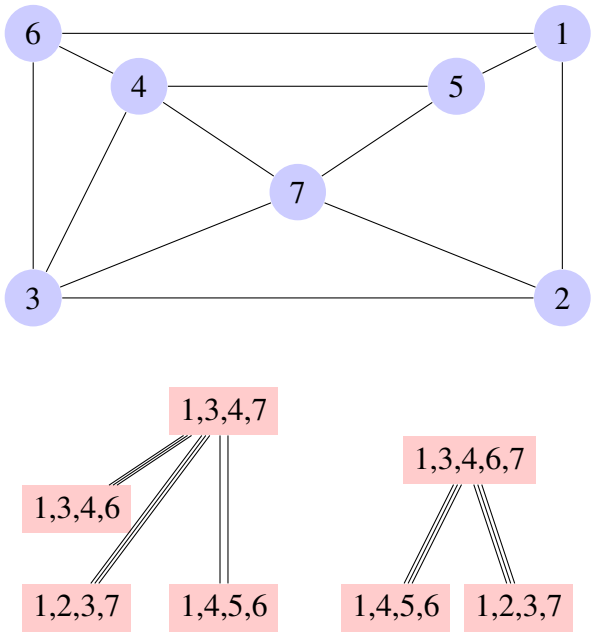


Fig. 1. A graph and two possible tree decompositions. One is minimal and the other is not. The graph has treewidth 3.

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 Bertelé, 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 significance 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 solve the 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 large constant described in the literature as "horrendous" [10]. Therefore, Bodlaender's algorithm is generally considered impractical for implementation [11]. Perkovíc and Reed gave an improvement to Bodlaender's algorithm with a recursion depth of up to k^2 [14]. However, recursion depth of k^2 results in $O(k^2)$ calls to an exponential-time base case [10].

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 simplicial in the subgraph formed by all vertices from i to n. A vertex is simplicial if its neighbours induce a clique. Algorithms for generating perfect elimination orderings also serve as heuristics for treewidth upper bounding.

One heuristic based on simplicial vertices and elimination orderings is the minimum degree heuristic [12]. In this heuristic, choose a vertex v of minimum degree, introduce edges such that v is simplicial. 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 tighten the upper bound for treewidth much lower than the trivial upper bound of $n - 1$.

The algorithmic focus of this report is on exact algorithms for treewidth. The standard algorithm for determining treewidth was given by Bodlaender in 2012 [6]. The algorithm uses dynamic programming and operates similar to the dynamic programming solution for the Traveling Salesman Problem given by Held and Karp. The algorithm begins with an empty decomposition 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 when computing treewidth. The algorithm uses the fact that treewidth is the *maximum* cardinality of the *minimum* tree decomposition. The definition results in a principle of optimality as needed for a dynamic programming algorithm. 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 a graph. Since the treewidth problem is NP-Complete and finding a tree decomposition is NP-Hard, much work has been accomplished to find appropriate solutions to the treewidth problem in appropriately limited scopes. These solutions 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 an 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 of treewidth was recognized by both Bertelé and Robertson and Seymour. However, no progress was made in determining the algorithmic benefits of treewidth until Bodlaender explored the usage of bounded treewidth for dynamic programming [16]. 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 solutions 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 [15]. 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 Three-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. Three-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. □

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 [17]. 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. □

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 [18], *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 expression. 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. Using tree automata, M can be linear-time translated into a sentence M' such that D satisfies M' if and only if G satisfies M. The proof is given in detail as part of [18]. □

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 constants 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 result 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 treewidth 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 implementation-practical approaches in the literature for determining the treewidth of arbitrarily sized or structured graphs. The motivation of this work is to (1) highlight the discrepancy between the theoretical benefits and practical approachability of the treewidth property, and (2) develop efforts towards the reconciliation of this discrepancy. 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 by the development of efficient algorithms for determining treewidth and tree decompositions. The theoretical benefits of the treewidth property arise in graphs of upper-bounded treewidth. Therefore, the problem of principal interest is treewidth upper bounding. Lower bounds on $tw(G)$ are explored in [19] and [20]. 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, having first been discovered by Bodlaender [6].

A. The fixed-parameter treewidth algorithm

Bodlaender's fixed-parameter linear-time algorithm relies on the fact that treewidth is minor-closed. Let G be a graph and H a 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 has 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 friendly when 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 simplicial 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 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. Algorithm 1 describes an implementation for computing and contracting the maximal matching of G.

The maximal matching of a graph is a matching that cannot be grown to accommodate additional edges. A greedy algorithm is therefore sufficient to compute the maximal matching of G. To contract

Algorithm 1 Generate a Maximal Matching of G

```
1: procedure MAXIMALMATCHING(G)
2:     M ←LinkedList
3:     visited ← [0]*|V|
4:     for u ∈ V do
5:         if u not visited then
6:             visited[u] = 1
7:             for v → 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
```

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 adjacencies of v. The adjacencies are then filtered for duplicates since u and v may share 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.

Since treewidth is a minor-closed property, 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, D can be found in linear time.*

proof: The proof is given by Bodlaender in [21]. 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 producible 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 simplicial vertex removal case is applied.

2) Simplicial 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 simplicial vertex in G_i. These vertices are described as *i-simplicial* as they are simplicial in G_i but are not necessarily simplicial in G. The approach in this case is analogous to the minimum degree heuristic in [12] based on perfect elimination orderings. Simplicial vertices are a key aspect of tree decompositions as their neighbours will always be found in a single bag. Algorithm 2 describes an approach to generating G_i and subsequently generating G' from G.

The three procedures together form the complete second case of the algorithm. The improved graph is generated, and its simplicial vertices are pruned. Recall that simplicial vertices are vertices whose neighbours induce a clique. G_i can be computed from G in linear time. In its given implementation, the computation of G' from G_i does not run in linear time. Let C be a set which is at first empty and let V be the vertices constituting a possible clique. If the current vertex in V has every member 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-simplicial vertices from G gives a graph with a tree decomposition upper-bounded by $k-1$. It must first be noted that if any i-simplicial 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.

Algorithm 2 Generate G' from G

```
1:  procedure ImproveGraph(G)
2:      for u ∈ V do
3:          for v ∈ V − u do
4:              L_u = LowDegreeAdj(u)
5:              L_v = LowDegreeAdj(v)
6:              if |L_u ∩ L_v| ≥ k + 1 then
7:                  Add edge uv to G
8:              end if
9:          end for
10:     end for
11: end procedure
12:
13: procedure RemoveSimplicals(G)
14:     for u ∈ V do
15:         if Clique(Adjacencies[u]) then
16:             Delete vertex u from G
17:         end if
18:     end for
19: end procedure
20:
21: procedure Clique(VertexSet)
22:     clique ← []
23:     for u ∈ VertexSet do
24:         for v ∈ clique do
25:             if u ↛ v then
26:                 return False
27:             end if
28:         end for
29:         clique ← u
30:     end for
31:     return True
32: end procedure
```

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 decomposition of G in linear time with treewidth no greater than k.*

proof: D' is only missing the i-simplicial vertices (as they were previously deleted). Since each i-simplicial 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-simplicial 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. □

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 dependence 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-simplicial 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. Downey and Fellows note that the recursion in Bodlaender's 1992 algorithm is as deep as k^8 [11]. 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 overall. Apart from concerns relating to recursion, the algorithm still depends on a doubly exponential constant in terms of k [11]. The constant parameter to the fixed-parameter algorithm is extremely large. The algorithm quickly becomes impossible to execute for values of k greater than 2.

Despite its relatively simple nature, there are hidden costs to the standard treewidth algorithm that result in most implementation attempts (including the attempt undertaken as part of this work) to be prohibitively unfeasible. In particular, the algorithm relies on a recursive depth as high as $O(k^8)$, and a doubly exponential constant in terms of k [11]. Therefore, additional work must be undertaken to approach the treewidth problem from an algorithmic perspective.

The algorithm originally proposed by Bodlaender relies on an exponential-time base case. When the graph is sufficiently small, a finite or brute force algorithm is used. The most efficient algorithm to date is Bodlaender's 2012 dynamic programming

algorithm, which runs in $O(2^n)$ 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 [22], leaving this algorithm to remain as the standard exact approach. Bodlaender's 2012 algorithm is considered to be the state of the art 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 functions in a manner 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 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: 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 candidate bag are $v \cup S$. If the number of reachable vertices to $v \cup S$ 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. The number of reachable vertices from $v \cup S$ can be determined via depth first search. A pseudocode implementation of this stage is given in algorithm 3.

Since the reachable vertices will form a single bag, the width of the tree decomposition (at this point) is the maximum size between the previous tree decomposition and the total number of reachable vertices. If the candidate decomposition width exceeds the existing upper bound, the decomposition is discarded and the next bag in the current decomposition is considered. Each vertex is evaluated in a for loop, with a tree decomposition being built to account for all vertices up to and including the current vertex under consideration.

Algorithm 3 DFS For Reachable Vertices

```
1:  procedure BUILDDECOMPOSITION(G, S, v)
2:      p ← width of previous decomposition
3:      r' = GETREACHABLETOTAL(G, S, v)
4:      r = max(r, p)
5:      if r ≤ up then
6:          // Update Width or introduce bag S ∪ v
7:      else
8:          // Skip to next bag S+
9:      end if
10: end procedure
11:
12: procedure GETREACHABLETOTAL(G, T, v)
13:     total ←0
14:     visited ← []
15:     S ← Stack
16:     push v onto S
17:     while S not empty do
18:         found ← False
19:         u ← S.peek()
20:         for w → u do
21:             if w unvisited then
22:                 found ← True
23:                 break
24:             end if
25:         end for
26:         if found then
27:             mark w visited
28:             if w ∉ S then
29:                 total++
30:             else
31:                 S.push(w)
32:             end if
33:         else
34:             S.pop()
35:         end if
36:     end while
37:     return total
38: end procedure
```

Once every vertex is considered, the algorithm stops and the width is determined.

2) Stopping Conditions: The algorithm relies on properties related to max cliques for early-exit optimizations. The first of these optimizations is given in theorem 2. The second optimization is as follows. Let C be the maximum clique in G. Then $tw(G)$ = max($tw(V - C)$. $|C| - 1$). The proof is given in [6]. Using these optimizations, the authors are able to exclusively consider vertices not included in the maximum clique to generate the tree decomposition. The minimal tree decomposition is generated by evaluating all non-clique vertices, and its width is returned as the final result.

Pseudocode for the algorithm is given in its original publication [6]. A C implementation of the algorithm including code for finding a maximum-clique is completed as part of this work and is available at [23]. Determining the maximum clique in G is NP-Complete [24]. However, an exponential time algorithm will not affect the run time of the primary algorithm, as it runs in exponential time in the first place. Unlike Bodlaender's linear time algorithm, this algorithm is practical for implementation and has been successfully completed by the authors themselves and as part of this work. Despite its relative ease of implementation, the performance of the algorithm does not effectively scale to graphs of large sizes. The results of this algorithm will be examined in section VII.

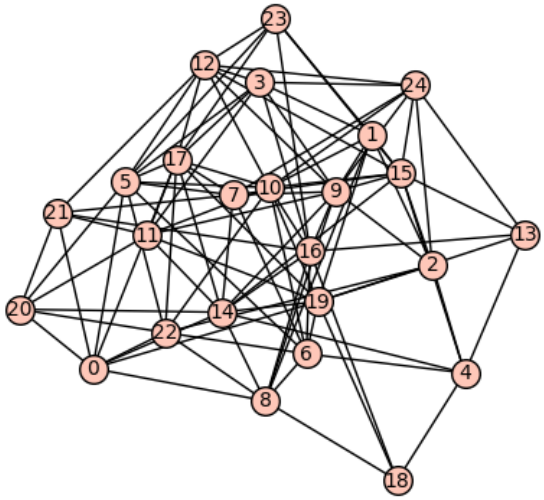
VI. Experimental Setup

The experimentation stage of this work concerns the performance comparison between a C implementation of Bodlaender's 2012 treewidth algorithm and the treewidth algorithm available on Sage, a popular mathematical software package [7]. Experiments were conducted on an eight-core Intel(R) i7-4720 processor with 3.35GHz speed and 8 GB of memory. The results are intended to compare the performance of both implementations and test their respective limits. The algorithms were executed against a series of small graphs, followed by three larger graphs. These graphs were the Dodecahedron, the Thomassen-34, and the Grid graph on 5 vertices. Additionally, a 25-vertex random graph was used. Table 1 enumerates the full set of inputs used.

TABLE I
Input Graphs

Name	Vertices	Edges	Treewidth
K_4	4	6	3
K_5	5	10	4
K_6	6	15	5
$K_{3,3}$	6	9	3
$K_{4,5}$	9	20	4
$Wheel_6$	6	10	3
$Wheel_7$	7	12	3
$Path_3$	3	2	1
$Path_5$	5	4	1
$SpanningTree$	14	13	1
$Cube_3$	8	12	3
$Cube_4$	16	32	6
$Petersen$	10	15	4
$Dodecahedron$	20	30	6
$Thomassen34$	34	52	4
$Grid_{5,5}$	25	40	5
$Random25$	25	103	12

Fig. 2. Randomly generated graph of 25 vertices



The majority of the inputs are small in size. Some of their tree decompositions could likely be determined by hand. However, there are also many graphs whose tree decompositions must be determined algorithmically. The intension of these benchmarks is to (1) test both algorithms for correctness, (2) determine the speed of these algorithms in an ideal case, and (3) give both algorithms inputs expected to be difficult. Overall, it is determined that the algorithm in Sage fares better than the tested implementation of Bodlaender's 2012 algorithm. However, this does not mean that the Sage implementation is without its own issues.

VII. Experimental Results

The results of experiments are divided into those concerning small, large and randomized inputs. While only three large graphs and only one random graph are used, they provide a sufficient picture of how the two algorithms compare. The treewidth implementation available in Sage appears to generally outperform the dynamic programming approach, with some notable exceptions.

A. Small Graphs

The small graphs tested produced nearly identical results between the two implementations. For the purpose of these experiments, all graphs excluding the final four are considered "small". The dynamic programming and Sage approaches were able to compute the treewidth of nearly all graphs in under 1 second. The graph $Cube_4$ was processed by both implementations in approximately 6 seconds. The only difference in performance in this set of inputs was on the graph $SpanningTree$. $SpanningTree$ was processed instantaneously by Sage, whereas the dynamic programming approach took approximately 3 seconds. However, since the number of edges in a tree on n vertices is always $n - 1$, the tested implementation of Bodlaender's 2012 dynamic programing algorithm is able to meet the benchmark set by the Sage implementation using a preprocessing step.

B. Large Graphs

The results from large graphs heavily favour Sage over the implementation completed in this work. The "large" graphs in this experiment are the graphs $Thomassen34$, $Dodecahedron$ and $Grid_{5,5}$. Sage was able to compute the treewidth of these graphs in 4, 14, and 13 seconds respectively. Meanwhile, the dynamic programming algorithm required multiple hours of execution to determine the treewidth of the same graphs. After six hours, only $Dodecahedron$ had been completed. $Thomassen34$ and $Grid_{5,5}$ did not complete execution within six hours.

Figure 3 highlights the time spent executing each graph for each vertex added to the tree decomposition. As mentioned, the dynamic programming algorithm adds one vertex at a time to the decomposition via a for loop. The exponential nature of the dynamic programming algorithm has a clear effect on its performance. As the decomposition increases in size, execution time rapidly increases. The increase in execution time is due to the requirement of the

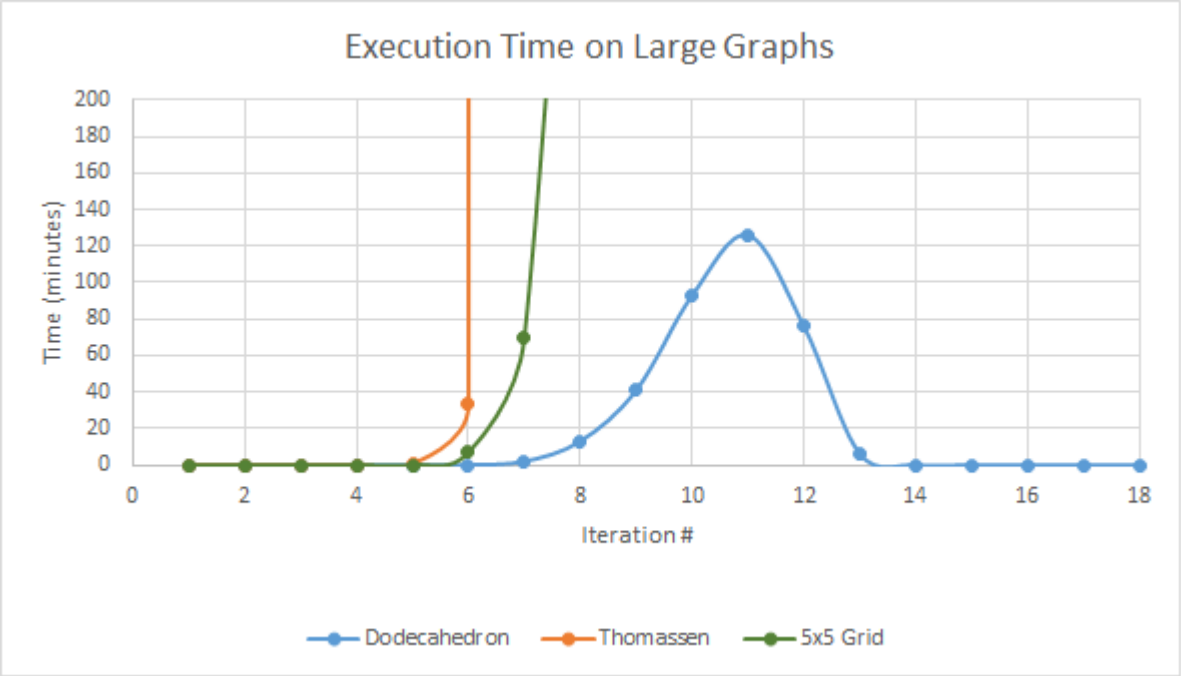


Fig. 3. Dynamic Programming Performance against Large Graphs

algorithm to inspect all past tree decompositions and their bags.

Once a solution is found, the algorithm is able to break out of the exponential complexity. Potential solutions with a higher treewidth than the upper bound are discarded. This is why the Dodecahedron input speeds up after iteration 12. If the optimal solution is not yet discovered, the runtime increases rapidly. $Thomassen34$ only completed six iterations. It's seventh iteration ran for over 24 hours and was thus cut off. Similarly, $Grid_{5,5}$ ran quickly until its seventh iteration. The graph was able to complete its eighth iteration in approximately 10 hours, which is outside the bounds of realistic performance in the context of these experiments.

Whereas the dynamic programming algorithm could only complete one of three large graphs in six hours, the Sage implementation was able to compute all three treewidths in under one minute. A likely cause for the advantage of Sage over the dynamic approach is the optimizations being used by the Sage implementation. The Sage algorithm first computes the cut vertices of G before computing the treewidth. The blocks are then computed for their treewidth separately. The bags of the blocks are then joined with a new bag that accounts for the edges of the cut vertex. As a result, graphs with multiple blocks can be computed much faster in Sage than treewidth. However, the advantage of the Sage algorithm disappears in graphs without a cut vertex.

C. Random Graph

The final comparative test undertaken between the two graphs concerns the random graph shown in figure 2. The importance of $Random25$ is that it has no cut vertices. Therefore, any symmetry-dependent strategy undertaken by the Sage implementation will not be useful on this input. As expected, this particular input results in the dynamic programming approach computing the treewidth *faster* than the Sage approach. The dynamic programming approach implemented was able to determine the treewidth of $Random25$ in 3 hours, 4 minutes and 42 seconds. On the other hand, the Sage implementation was unable to compute the treewidth after 5 hours. At this point, the Sage implementation was cut off.

The random graph benchmark shows that the treewidth algorithm available in Sage is not necessarily better than the dynamic programming approach developed by Bodlaender in [6]. However, the Sage implementation does highlight the usefulness of various optimization strategies and preprocessing steps. If the dynamic programming algorithm were to handle known results such as those concerning the treewidth of complete graphs and trees, the algorithm may see a performance gain. However, even with clever optimizations including cut-vertex handling, algorithms for treewidth do not practically scale to large graphs in their current state. The "best" result of the dynamic programming approach - a graph of 25 vertices - ran for multiple hours. Such enormous execution times highlight a clear need to continue the pursuit of optimizations, heuristics and algorithms towards efficient treewidth and tree decomposition approaches.

VIII. Conclusions and Future Work

This work compared two noteworthy algorithms for computing the treewidth of a graph. The first was Bodlaender's 1992 fixed-parameter algorithm. The algorithm was deemed too impractical for implementation due to its prohibitive dependence on large constants in its runtime. The algorithm's recursive depth and specific data structure needs also contribute to its impracticality.

The second algorithm was Bodlaender's 2012 dynamic programming algorithm. This algorithm is much more practical for real-world use, and has been implemented successfully both in its original publication and as part of this work. A variety of graphs were used to compare the dynamic programming algorithm against the treewidth algorithm provided by Sage. The two algorithms performed similarly for the vast majority of graphs. However, large graphs were computed faster by Sage except for a key exception. A random graph with very little symmetry was computable by the dynamic programming approach, and could not be computed by Sage in a timely manner. The advantage of the dynamic programming algorithm over Sage in this case arises from the Sage implementation's dependence on symmetries such as cut vertices. Despite

relative performance differences, neither implementation was able to adequately handle graphs of more than 20 vertices.

The failure of the treewidth algorithms to handle reasonably-sized graphs motivates two aspects of future work. First, efforts must be undertaken to further improve the existing state of the art in treewidth algorithms. Optimizations, heuristics and alternative approaches must be explored if the treewidth property is to be attainable for graphs on more than very few vertices. The second aspect of future work concerns the theoretical categorization of the treewidth problem. The impracticality of the presented fixed-parameter treewidth algorithm leads to the notion that the fixed-parameter tractable complexity class may not be a true part of the polynomial-time category. The fixed-parameter tractability of the treewidth problem and similar problems motivates extensive development into a new type of system of complexity theory, suitably named *parameterized complexity*.

REFERENCES

[1] Robertson, Neil, and Paul D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. Journal of algorithms 7.3 (1986): 309-322.

[2] Bodlaender, Hans L. Dynamic programming on graphs with bounded treewidth. International Colloquium on Automata, Languages, and Programming. Springer, Berlin, Heidelberg, 1988.

[3] Robertson, N., & Seymour, P. D. (1984). Graph minors. III. Planar tree-width. Journal of Combinatorial Theory, Series B, 36(1), 49-64.

[4] Bodlaender, H. L. (2006, June). Treewidth: characterizations, applications, and computations. In International Workshop on Graph-Theoretic Concepts in Computer Science (pp. 1-14). Springer, Berlin, Heidelberg.

[5] Bodlaender, H. L. (1996). A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM Journal on computing, 25(6), 1305-1317.

[6] Bodlaender, H. L., Fomin, F. V., Koster, A. M., Kratsch, D., & Thilikos, D. M. (2012). On exact algorithms for treewidth. ACM Transactions on Algorithms (TALG), 9(1), 12.

[7] SageMath, the Sage Mathematics Software System (Version 8.1), The Sage Developers, 2018, http://www.sagemath.org.

[8] Bertele, U., & Brioschi, F. (1972). Nonserial dynamic programming. Academic Press.

[9] Arnborg, S., Corneil, D. G., & Proskurowski, A. (1987). Complexity of finding embeddings in ak-tree. SIAM Journal on Algebraic Discrete Methods, 8(2), 277-284.

[10] Bodlaender, H. L. (2005, January). Discovering treewidth. In International Conference on Current Trends in Theory and Practice of Computer Science (pp. 1-16). Springer, Berlin, Heidelberg.

[11] Downey, R. G., & Fellows, M. R. (2016). Fundamentals of parameterized complexity (Vol. 201, No. 3). Springer.

[12] Bodlaender, H. L., & Koster, A. M. (2010). Treewidth computations I. Upper bounds. Information and Computation, 208(3), 259-275.

[13] Paul D. Seymour. Online Correspondence. https://cstheory.stackexchange.com/questions/5018/the-origin-of-the-notion-of-treewidth/5020. Last Accessed April 2018

[14] Perkovi, L., & Reed, B. (1999, June). An improved algorithm for finding tree decompositions of small width. In International Workshop on Graph-Theoretic Concepts in Computer Science (pp. 148-154). Springer, Berlin, Heidelberg.

[15] Arnborg, S., & Proskurowski, A. (1989). Linear time algorithms for NP-hard problems restricted to partial k-trees. Discrete applied mathematics, 23(1), 11-24.

[16] Bern, M. W., Lawler, E. L., & Wong, A. L. (1987). Linear-time computation of optimal subgraphs of decomposable graphs. Journal of Algorithms, 8(2), 216-235.

[17] Robson, J. M. (2001). Finding a maximum independent set in time O (2n/4). Technical Report 1251-01, LaBRI, Universit Bordeaux I.

[18] Courcelle, B. (1990). The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. Information and computation, 85(1), 12-75.

[19] Bodlaender, H. L., & Koster, A. M. (2011). Treewidth computations II. Lower bounds. Information and Computation, 209(7), 1103-1119.

[20] Gogate, V., & Dechter, R. (2004, July). A complete anytime algorithm for treewidth. In Proceedings of the 20th conference on Uncertainty in artificial intelligence (pp. 201-208). AUAI Press.

[21] Bodlaender, H. L., & Kloks, T. (1991, July). Better algorithms for the pathwidth and treewidth of graphs. In International Colloquium on Automata, Languages, and Programming (pp. 544-555). Springer, Berlin, Heidelberg.

[22] Downey, R. G., & Fellows, M. R. (2013). Heuristics for Treewidth. In Fundamentals of Parameterized Complexity (pp. 205-211). Springer, London.

[23] Github page for COMP 4060 Project by Taylor Cox. https://github.com/CoxT9/GraphTheory-Project. Accessed April 2018

[24] Karp, R. M. (1972). Reducibility among combinatorial problems. In Complexity of computer computations (pp. 85-103). Springer, Boston, MA.

[25] Parameterized Complexity Community Webpage. http://fpt.wikidot.com/. Accessed April 2018