Solving Treewidth in graphs using MaxSAT

João Varandas
joao.varandas@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

November 2018

Abstract

Treewidth is an important property of graphs. Knowing the treewidth value for a given graph, and the corresponding optimal tree-decomposition, allows one to reduce and approximate well-known NP-Hard problems. This means that, there are NP-Hard problems which can be solved efficiently by approximating the problem instance to a tree-decomposition of small width. Despite the fact that knowing the treewidth value for graphs can be a solution for some NP-Hard problems, finding the treewidth value is an NP-Hard problem itself. In this document it is provided a survey on algorithms for solving the treewidth value of an arbitrary input graph with emphasis on the usage of SAT and MaxSAT formalizations. Moreover, it is also outlined a proposal to find the treewidth value of a graph, as well as the corresponding decomposition, using a MaxSAT based approach. Finally, the proposed solution is described, the evaluation the procedure is described, and the results are discussed.
Keywords: Treewdith, Tree-Decomposition, Linear Ordering, Maximum Satisfiability

1. Introduction

The treewidth of a graph is an important measure on graph theory. It was introduced by Robertson and Seymour[26] in 1986 and is one of the most relevant structural graph properties that measures what is the "distance" of some graph from being a tree. Smaller the treewidth, closer the graph is from being a tree. However, finding an optimal *tree-decomposition*, that is, the *tree-decomposition* with the lowest width, is an NP-hard problem[1].

Arnborg[2] sentenced that some NP-Hard problems over graphs, can be computed in linear or polynomial time when restricted to partial *k-trees*, that is, graphs with treewidth bounded to a known integer k. Examples of well known problems over graphs are: (1) the maximum independent set problemt; (2) the chromatic number problem (also known as the graph coloring problem); or (3) the Hamiltionian circuit. Other NP-Hard combinatorial problems exists related to the treewidth value of graphs. Such combinatorial problems in graphs have been studied for years, and several solutions have been developed to solve them in graphs with bounded treewidth[7, 23, 5, 17, 22].

To solve the treewidth problem, it is possible to recognize two main approaches. The exact ones, which has as the main objective identify the exact value for treewidth of a given graph. And the approaches based on approximations and heuristics, where the goal is to compute approximated values for the treewidth of a given graph.

Different fields on computer science are searching to address its problems by using the knowledge gathered on the tree-decompositions constructed for the input graphs. Examples of treewidth application in diverse science fields are logic, where treewidth is used to optimize algorithms on constraint satisfaction[21, 11, 15], artificial inteligence, where large graphs are processed to build bayesian networks[13, 4, 25, 20], computational biology[16, 18, 31, 32]

PACE challenge 2017 [1] is an annual competition that was organized to evaluate solutions to compute the treewidth of a graph. The challenge had two main challenges, where one of them was computing the treewidth of a given graph. Two variants for the challenge have been proposed. The first one evaluated who solved the most number of instances. And the second one, evaluated who computed the tree-decomposition of smaller width[12].

In this work we provide a solution to find the treewidth value of a given graph, and the respective optimal linear ordering, using a propositional logic based approach. To find the optimum value for the treewidth, we proposed to design a solution based in algorithms which use two main deciding stages, the pre-processing one, which computes an upper-bound and a lower-bound on the treewidth, and a MaxSAT based component, which use a MaxSAT encoding to decide the treewidth value.

[1] https://pacechallenge.wordpress.com

2. Background

A *graph* $G = (V, E)$ is a tuple where V is a non-empty set of vertices and $E \subseteq \{\{x, y\} | x, y \in V\}$ is a set of edges. Throughout the document, $V(G)$ is used to denote the set of vertices of the graph G, $E(G)$ is used to denote the set of edges of the graph G. The set of neighbors of a vertex v is denoted by $N(v)$. The degree of a vertex v, $degree(v)$, is the number of vertices connected to v through an edge $e \in E$, $|N(v)|$.

2.1. Treewidth

A *tree-decomposition* of a *graph* $G = (V, E)$ is a tree $T = (V', E')$ where each node $t \in V'$ is a set $\chi(t) \subseteq V$ that respects the following conditions:

1. $\bigcup_{t \in V'} \chi(t) = V$.
2. If $\{u, v\} \in E$ then $u, v \in \chi(t)$ for some node $t \in V'$.
3. For all nodes $t_1, t_2, t_3 \in V'$, if t_2 is on the path from t_1 to t_3 then is known that $\chi(t1) \bigcap \chi(t_3) \subseteq \chi(t_2)$.

The first rule empowers that the union of all sets $\chi(t)$ on the *tree-decompositions* of the graph G contains all the vertices of V. The second rule requires that a pair of vertices connected by an edge on G must be together on some set $\chi(t)$ of T. The third rule requires that if some vertex on V is in the sets of two different nodes on the tree, then the vertex occur in every node on the unique path between them on the tree.

The width of a *tree-decomposition* is given by $max_{t \in V'} |\chi(t)| - 1$. Where $max_{t \in V'} |\chi(t)|$ represents the largest set on the tree, i.e. the node of the tree that contains the biggest number of vertices of G.

The treewidth of an undirected *graph* G, $tw(G)$, is defined in terms of the *tree-decompositions* of G as the minimum width over all *tree-decompositions* of G.

The width of a *tree-decomposition* can also be defined in terms of some linear ordering $\prec$ of the vertices of the graph. Given a linear ordering $\prec$ of the vertices in V it is possible to define *successor* and *predecessor* such that:

- v_i is said to be a *successor* of v_j under $\prec$ if $\{v_i, v_j\} \in E$ and $i > j$.
- v_i is said to be a *predecessor* of v_j under $\prec$ if $\{v_i, v_j\} \in E$ and $i < j$.

To compute the width of the *tree-decomposition* respecting the linear ordering $\prec$ it is possible to define a triangulation $\Delta(G, \prec) = (V, \Delta(E, \prec))$, from which the corresponding width can be read off, without constructing the *tree-decomposition*. The edge-relation $\Delta(E, \prec)$ represents the edges resulting in the triangulated graph. An example on the mentioned concepts is given in figure 1.

2.2. SAT Based Approach

The Boolean satisfiability problem (SAT), was the first problem to be proven NP-complete[10]. The SAT problem consists on proving if exists a truth assignment that satisfies a given propositional logic formula in the conjunctive normal form, or prove that the formula is unsatisfiable.

A Boolean formula is said to be in conjuctive normal form (CNF), if is a conjunction of clauses. A clause is a disjunction of literals ($\vee$, logical OR). A clause is called a unit clause if it contains exactly one literal, or it is called a k-clause if it contains exactly k literals. A literal is a Boolean variable p or its negation, $\neg p$.

A truth assignment τ is a function from Boolean variables (literals) to $\{0, 1\}$. A clause C is satisfied by a truth assignment τ, which means that $\tau(C) = 1$ if and only if $\tau(p) = 1$ for a literal $p \in C$ or, if and only if $\tau(p) = 0$ for a literal $\neg p \in C$. Otherwise, the clause C is meant to be unsatisfiable, which means that $\tau(C) = 0$.

The maximum satisfiability problem (MaxSAT) is an optimization version of the SAT problem. Given a propositional logic formula φ in CNF with m clauses and an integer k such that $k \leq m$ the MaxSAT problem tries find a truth assignment to the variables of φ that satisfies at least k clauses in φ.

This section introduces the work that has been done on SAT encodings to solve the treewidth problem. In order to do that, we focus our studies on two papers of main relevance. The first is the work of Samer and Veith[29], and the second, is the work of Berg and Järvisalo[3] which includes a MaxSAT approach to compute the treewidth. Since the work of Berg and Järvisalo is based on the work of Samer and Veith, improving the SAT encoding, the work done by Berg and Järvisalo is described, as well as the main differences to the work previously done by Samer and Veith.

To encode the treewidth problem into SAT, Berg and Järvisalo use two sets of Boolean variables, which are manipulated in order to solve the problem. The Boolean variables used to produce the encoding are the following:

- $ord_{i,j}$, which is true if, and only if, the vertex v_i precedes the vertex v_j, that is, $v_i < v_j$ under the linear ordering $\prec$.
- $O_{i,j}$, which is true if, and only if, the ordered graph of G, under $\prec$, that is $\overrightarrow{\Delta}(G, \prec)$, contains the edge (v_i, v_j).

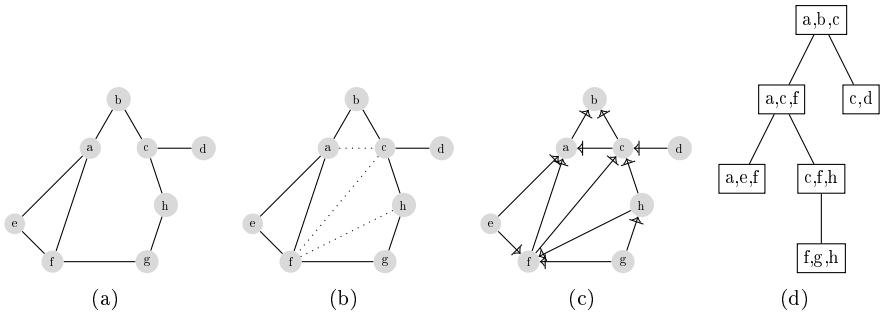

Figure 1: (a) An example of an undirected graph G of treewidth 2; (b) The triangulation $\Delta(G, \prec)$ under the linear ordering $\prec$: $g < h < e < f < d < c < a < b$; (c) The corresponding ordered graph under $\prec$; (d) A possible *tree-decomposition* of G with width 2. (Adapted from [28])

Firstly, the linear ordering itself, is encoded by the set of variables $ord_{i,j}$ for each $1 \leq i < j \leq n$, which implies to have $n(n-1)/2$ Boolean variables in the encoding. The transitivity property of the linear ordering is encoded by

$$(ord^*_{i,j} \wedge \ ord^*_{j,l}) \rightarrow ord^*_{i,l}, \tag{1}$$

where $ord^*_{p,q}$ is $ord_{p,q}$ if $p < q$, otherwise, $ord^*_{p,q}$ is $\neg ord_{p,q}$. This means that, if a vertex v_i precedes a vertex v_j, and v_j precedes a vertex v_l, under some linear ordering $\prec$, then, v_i precedes v_l under $\prec$. The encoding of the transitivity property implies the addition of $n(n-1)(n-2)$ constraints to the SAT instance.

Knowing that the input graph $G = (V, E)$ contains an edge $\{v_i, v_j\}$, then, the ordered graph $\overrightarrow{\Delta}(G, \prec)$ contains the directed edge (v_i, v_j), or the directed edge (v_j, v_i). This can be enforced by adding a set of clauses of the form

$$O_{i,j} \vee O_{j,i} \tag{2}$$

for all $\{\{v_i, v_j\} \in E : i < j\}$.

If $v_i, v_j \in V$ have a common predecessor in the graph $G = (V, E)$, then, processing the triangulation procedure, the resulting graph $\Delta(G, \prec)$, will contain the edge $\{v_i, v_j\}$. Thus, the ordered graph $\overrightarrow{\Delta}(G, \prec)$ contains the directed edge (v_i, v_j) or the directed edge (v_j, v_i). This is enforced by the following equation:

$$(O_{k,i} \wedge O_{k,j}) \rightarrow (O_{i,j} \vee O_{j,i}) \tag{3}$$

for all distinct $i, j, k = 1..n$.

Since the linear ordering leads to an ordered graph $\overrightarrow{\Delta}(G, \prec)$, in which the direction of each edge $\{v_i, v_j\} \in E$, is ruled by the linear ordering, considering that if $\{\{v_i, v_j\} \in E : i < j\}$ then $\overrightarrow{\Delta}(G, \prec)$ contains the edge directed edges (v_i, v_j), otherwise, $\overrightarrow{\Delta}(G, \prec)$ contains the edge (v_j, v_i) for all distinct $i, j = 1..n$. Knowing that, we must decide what is the direction of each edge $\{v_i, v_j\} \in E$. To decide the direction of each edge of the ordered graph, the authors introduce the formula:

$$ord^*_{i,j} \rightarrow \neg O_{j,i} \tag{4}$$

for all distinct $i, j = 1..n$, which means that, if $i < j$ under the linear ordering $\prec$, then, it can not exist a directed edge (v_j, v_i) in the ordered graph $\overrightarrow{\Delta}(G, \prec)$.

Finally, it is added a redundant clause, of the form

$$\neg O_{j,i} \vee \neg O_{i,j} \tag{5}$$

for all distinct $i, j = 1..n$, that improves the time required for solving the resulting SAT (and MaxSAT) instance, according to the authors. The clause is added since the fact that the ordered graph $\overrightarrow{\Delta}(G, \prec$) is a simple graph, which means that, if there is an edge (v_i, v_j) it can not exist an edge (v_j, v_i).

The conjunction of equation 1, with equations 2, 3, 4, 5, results in the base encoding. The SAT instance created by the base encoding, for a given graph G, is represented by $\varphi_{base}(G)$.

deciding treewidth

In order to decide which is the treewidth value for a given graph $G = (V, E)$, the authors use the notion of the maximum number of successors, given the triangulated graph $\Delta(G, \prec)$. This means that, recalling the fact that the treewidth of a tree-decomposition, corresponding to a linear ordering $\prec$ is given by the maximum number of successors in the triangulated graph, that is, $max_{v_i \in V} |\{\{v_i, v_j\} \in \overrightarrow{\Delta}(E, \prec) : i < j\}|$ then, this leads us to infer that the number of outgoing arcs of each vertex in the ordered graph $\overrightarrow{\Delta}(G, \prec)$, can not exceed the upper bound value for treewidth. Thus, to enforce this restriction, it is added a cardinality constraint of the form

$$C_w(i) = \sum_{\substack{j=1 \\ j \neq i}}^{n} O_{i,j} \leq k \tag{6}$$

for each $i = 1..n$. Let $C_w(i)$ denote the cardinality constraint for each vertex v_i.

Joining the base encoding $\varphi_{base}(G)$ with the equation 6, we have

$$\varphi_{base} \wedge \bigwedge_{i=1}^{n} C_w(i) \tag{7}$$

which is satisfiable if, and only if, $tw(G) \leq k$. Note that if equation 7 evaluates to SAT, the variables $ord_{i,j}$ represents the optimal linear ordering, that is, the linear ordering which allows one to compute the optimal *tree-decomposition*. Knowing the optimal linear ordering, a corresponding *tree-decomposition* of width at most k is constructed[6].

Hence, the treewidth of a given graph G, can be achieved by calling the SAT solver mutliple times, incrementing the value k, from 1 to $n-1$, until equation 7 is satisfied.

To encode equation 6, the authors use a clausal encoding based on cardinality networks, where $A_i = \{O_{i,j} : i \neq j\}$ is a set, from which the cardinality network $C(A_i)$ produce $|A_i| - 1$ auxiliary variables, $y_1^i..y_{n-1}^i$, which indicate how many variables in A_i are assigned to true. Note that the conjunction of the clauses produced by $C(A_i)$, with the clause $\neg y_{k+1}^i$ is equivalent to equation 6. Knowing that, it is possible to produce a SAT instance based on $C(A_i)$, which is true if, and only if, $tw(G) \leq k$, assuming that $\tau(y_{k+1}^i) = 0$ for each $i = 1..n$. Thus, the following equation is constructed:

$$\varphi'_{iter} = \varphi_{base}(G) \wedge \bigwedge_{i=1}^{n} C(A_i) \tag{8}$$

To enforce that $\tau(y_{k+1}^i) = 0$ for each $i = 1..n$, a new set of auxiliary variables W_i is constructed, where $i = 0..(n-2)$, such that

$$W_i \leftrightarrow (\neg y_{i+1}^1 \wedge \neg y_{i+1}^2 \wedge .. \wedge \neg y_{i+1}^n) \tag{9}$$

Hence, the treewidth value can be achieved by finding the smallest value k, for which the SAT instance, composed by the logical conjunction of equations 8 and 9, is satisfiable if, and only if $W_k = 1$. Note that when $W_{n-2} = 1$, it is possible to infer that $tw(G) = n - 1$.

Finally, the authors insert a clause that is based on the notion that, for any k, knowing that $tw(G) \leq k$, then, $tw(G) \leq k'$ for all $k' > k$. This is added to the SAT instance, constructing formulas of the form

$$W_i \rightarrow W_{i+1} \tag{10}$$

for $i = 0..(n-3)$.

The logical conjunction of equations 8, 9, 10 is called the *iterative encoding* by the authors, and it is denoted by $\varphi_{iter}(G)$. That means that $\varphi_{iter}(G)$ can be solved iteratively, under different assumptions, allowing the solver to retain information between iterations, avoiding to rebuild the SAT solver in each iteration.

The most relevant difference between the work done by Samer and Veith[29], and the work done by Berg and Järvisalo[3], consists on the encoding of equation 6. The former uses a sequential counter to enable the treewidth computation, using multiple calls to the SAT solver given different values to the upper bound k. The last uses an iterative encoding that can be solved using an incremental setting of a SAT solver, enabling the solver to learn information between iterations. Both base encodings use the same ideas. However, since the fact Berg and Järvisalo use the notion of directed edges, instead of the notion of successors, it allows one to abstract the triangulation process on the input graph G, when Samer and Veith explicitly construct constraints to represent the edges added on the triangulation.

3. Implementation

To solve the treewidth problem, we segment the implementation into three main stages. The first stage consists on a pre-processing approach, which allows one to bound the treewidth search space. The pre-processing level is constructed by implementing lower and upper bound algorithms, to verify if they coincide. In case the lower bound equals the upper bound, then, we achieved the exact value for the treewidth. Otherwise, we search by the exact treewidth value between the lower and upper bounds.

Another level on the implemented software for search the exact treewidth value, we called the base encoding. The base encoding is common to every algorithm implemented, and it is the level where the problem is encoded into a SAT instance. Encoding the equations described in section 2 into SAT, a propositional logic based formula is produced, to enforce rules that drive the search algorithms to the correct result on the treewidth.

Finally, we construct multiple algorithms to search the exact treewidth value. These algorithms are bounded to search in the space that exists between the lower and the upper bound. Moreover, the algorithms make use of the SAT formula, ϕ, produced by the base encoding, at each step of the search procedure, to decide whether the correct treewidth value was found. These search algorithms use multiple calls to a SAT solver, in order to understand whether the rules encoded in ϕ are satisfied. In each call, the SAT solver returns a tuple, containing information about the *status*, the *model*, and the *conflicts*. The *status* gives information on the satisfiability of the SAT instance ϕ, being set to

SAT in the case of the SAT instance ϕ is satisfiable, or is set to UNSAT, if the SAT instance is unsatisfiable. The *model* is the model returned by the SAT solver, which is empty in case of the *status* is set to UNSAT, or gives the variable assignment that satisfies all the constraints. The *conflicts* parameter is an array which is empty in case of the *status* is SAT, otherwise, it gives an unsatisfiable core.

Building the software to solve the treewidth problem over arbitrary graphs, we used the architecture of the OpenWBO MaxSAT solver[24], which achieved two gold medals and a silver medal in the MaxSAT evaluation 2017[2]. The architecture of the software developed creates an abstraction on the mentioned levels of the solution, making the algorithms for search the treewidth to be independent on the pre-processing algorithms, using these algorithms by making a simple call. Moreover, it encapsulates the implementation of cardinality encoding algorithms, which facilitates the CNF encoding of equation 6.

Implementing lower bound algorithm, we base our implementation on the Minor-Min-Width algorithm. This algorithm is based on computing minors, taking advantage of the use of edge contractions to choose the maximal minimum degree, at each step of the algorithm. This algorithm outperforms other known algorithms for computing lower bounds, both in terms of the quality of the results, as in terms of the running time of the algorithm, for the majority of the cases.

To compute the upper bound algorithm on treewidth, we implemented an improved version of the Greedy Fill-In algorithm. Greedy Fill-In algorithm was chosen since previous experiments on upper bound algorithms to treewidth shows that Greedy Fill-In may be the best algorithm to computing upper bound values on treewidth, both in terms of the quality of the solution returned, as the running time of the algorithm[8]. Moreover, we join a minimum-degree based heuristic to the algorithm, which is slightly better than the original greedy fill-in, following Bodlaender and Koster[9]. Furthermore, we find a tweak to improve the performance of the implemented upper-bound algorithm. This improvement can be observed using the fact that, since the upper bound value is given by the maximum degree of a vertex v in V, at each step of the algorithm, then, it is not possible to increment the upper bound value when $|V| < ub$. Thus, to return an upper bound value for the treewidth, it is possible to stop searching for an upper bound when the upper bound value obtained is greater or equal than the number of vertices remaining in the graph.

Using SAT search algorithms for treewidth consists essentially on making calls to the SAT solver, and then making decisions on the returned result. Such algorithms are responsible for guide the decision process on the treewidth, deciding at each step of the algorithm whether the exact value for the treewidth of the input graph is found, or if it is necessary to adjust the parameters to produce a new call to the SAT solver, until the optimal value is achieved. Regarding the implemented algorithms, we describe two specialized SAT based algorithms designed for search the treewidth value.

[2] http://mse17.cs.helsinki.fi/

Algorithm 1: Selected Unsatisfiability-based algorithm for Treewidth

Input: $G = (V, E)$
Output: treewidth value

```
1  ub ← Minimum Fill-In(G)
   lb ← Minor-Min-Width(G)
   φ ← TreeWidthEncode(G, ub) A ← ∅
2  foreach i ∈ 1...|V| do
3      Let (b_{i,1} ... b_{i,ub}) be the variables defining
        the LHS value of cardinality constraint
4      A ← A ∪ {¬b_{i,lb+1}}
5  while true do
6      (status, ν, φ_C) ← SAT(φ, A)
7      if status = UNSAT then
8          min ← ub
9          foreach ¬b_{i,j} ∈ φ_C do
10             if j < min then
11                 min ← j
12         foreach ¬b_{i,min} ∈ φ_C do
13             A ← A\{¬b_{i,min}} ∪ {¬b_{i,min+1}}
14     else
15         return ν
```

3.1. SelectedUS Algorithm

Selected Unsat-SAT algorithm (SelectedUS) starts by computing the lower and upper bounds, stopping when the values coincide. Then, the initial SAT instance is produced, bounding the number of outgoing arcs to the computed upper bound. The SAT instance produced is then modeled throughout the decision process of the algorithm.

Produced the initial SAT instance, the Selected Unsat-SAT algorithm, described in 1, constructs a vertex of assumptions, where the auxiliary variables $b_{i,j}$ used to encode the left-hand-side of the cardinality constraint 6 are bounded to the actual computed lower bound.

After the assumptions vertex initialization, the SAT solver is called over the produced formula set, and the assumptions array, returning a tuple from

which is possible to read the *status*, the *model*, in case it is not empty, or the *conflicts* vector, which contains the variables that are responsible for the SAT instance to be unsatisfiable.

If the *status* is set to UNSAT, then, a minimum value is initialized with the upper bound value on the treewidth, computed at the beginning of the algorithm. Then, the conflicts vector (which can not be empty, since the evaluation returned the status as UNSAT), is scanned, updating the minimum value for the minimum right-hand-side found in the conflicts vector.

Since we updated the minimum value, it is possible to update all the variables in the conflicts vector that are bounded to the previous minimum value, to be restricted to the previous minimum value plus one, which avoids to update all the variables in the conflicts, and consequently, avoids to update all the variables in the encoded SAT instance, as it was done in the other algorithms. Such a mechanism allows one to manipulate directly the SAT instance encoded, selecting precisely the variable which will be updated, in order to continue the search for the treewidth.

When the SAT instance is satisfied, we know that the exact value for the treewidth of the input graph was found. The exact value on the treewidth of the input graph corresponds to the maximum j for which the variables $b_{i,j}$ satisfies the SAT instance, thus, the algorithm outputs that value and stops.

SelectedUS algorithm is implemented over the incremental SAT solving, since the SAT instance is constructed and manipulated over each iteration of the algorithm, which allows to select and manipulate specific variables in the encoding, at each iteration of the algorithm.

Algorithm 2: Iterative algorithm for Treewidth

Input: $G = (V, E)$
Output: treewidth value

```
1  (ub, pr, ≺) ← Minimum Fill-In(G)
   lb ← Minor-Min-Width(G)
   c ← MIN(FREEVALUE, |V|)
2  A ← ∅
3  while true do
4      (φ, A) ← PartialEncode(G, ub, ≺, c, pr)
5      (status, ν, φ_C) ← SAT(φ, A)
6      if status = SAT then
7          c ← MIN(FREEVALUE, |V|)
8          ≺← UpdateOrdering(ν)
9          pr ← UpdatePointOfRegret(ub)
10         while pr = NIL do
11             ub ← ub − 1
12             if ub = lb then
13                 return ν
14             pr ← UpdatePointOfRegret(ub)
15     else
16         if φ_C ∩ A = ∅ then
17             return ν
18         c ← c+FREEVALUE
```

3.2. Local Improvement Algorithm

Local Improvement algorithm (LocalImprov), described in 2, consists in arranging an approximated to optimal linear ordering on the treewidth, and iteratively rearranging it until an optimal linear ordering is achieved. To enable such a feature, it is necessary to define a point of regret, which is defined as the index of the first vertex that makes the linear ordering to achieve a given value of successors (in this particular case, the point of regret is the index of the first vertex that represents the reason for the actual maximal number of successors of the treewidth - that is, the index of the vertex that defines the upper bound value on the treewidth value for a given linear ordering). Defined the point of regret, it is verified if there is a permutation of the point of regret and its c neighbors in the linear ordering, such that the actual upper bound value on the treewidth can be reduced.

The algorithm operates through an improved version of the base encoding. The main differences between the base encoding and the improved encoding is the way of produce the CNF encoding of equations 1 and 4, and the introduction of assumptions in the CNF encoding process. To encode the linear ordering, the vertices between the first position in the linear ordering, and the left bound computed at the beginning of the algorithm, are fixed, that is, it is assumed that the position of the vertices in the mentioned interval will not permute. The same happens to the vertices in the interval between the right bound computed at the start of the algorithm, and the end of the linear ordering. To enable the SAT solver to fix the position of these vertices, a vector of assumptions is constructed. To build the vector of assumptions, the set of variables $Ord_{i,j}$ is added to the vector, for all $i, j : i < j$ where $i = 1, ..., l-1$ and $j = i+1, ..., | \prec |$, which represents that the first l vertices of the linear ordering $\prec$ appears before any other vertices in $\prec$, being l the left bound of the linear ordering, computed at the beginning of the algorithm. Furthermore, the set of variables $Ord_{i,j}$ where, all $i, j : i < j$ such that $i = l, ..., r$ and $j = r+1, ..., | \prec |$ are added to the vector of assumptions, since the last $| \prec | - r$ always

appears after the vertices in the interval between l and r, being r the right bound, computed at the beginning of the algorithm. Finally, it is encoded the set of variables $Ord_{i,j}$ where all $i, j : i < j$ such that $i = r+1, ..., | \prec |$ and $j = i+1, ..., | \prec |$, which represents the last fixed vertices in the linear ordering. Equation 1 encoded into CNF, but only in the interval that exists between l and r.

LocalImprov algorithm starts by computing an initial linear ordering, which is given by the upper bound algorithm. Processing the computed linear ordering, is possible to infer an upper bound value for treewidth, as well as the point of regret. Then, the lower bound is computed by calling the lower bound algorithm. Additionally, it is necessary to define the value c, which represents a the bounds of the linear ordering that defines the interval where the vertices are allowed to permute. The value c is defined as the minimum value between the minor liberty level (which we call *FREEVALUE*)), and the size of the linear ordering. Since the interval on which the vertices are allowed to permute can not exceed the size of the linear ordering.

Defined all the variables, LocalImprov algorithm produces a verification to check if the lower and upper bound values coincides. If the values coincides, the algorithm returns the treewidth value and stops. Then, the improved encoding produces the formula set, as well as the vector of assumptions. Then, it is produced a call to the SAT solver, over the formula set and the assumptions vector.

Calling the SAT solver, the algorithm identifies the value stored in *status*. If the *status* returned by the SAT solver is set to SAT, then, the c value is restored to the initial value of c, in order to bound the new search that is performed at each new iteration of the algorithm. Furthermore, the actual linear ordering is updated to the one that is given by the Ord variables and its values in the model ν.

After updating the linear ordering, the algorithm updates the actual point of regret. Since the linear ordering was updated, then, it is possible that the point of regret changes too, so, it is needed to update the point of regret to a new one in case of it exists, for the actual upper bound. To update the point of regret, it is needed to verify which is the vertex in the updated linear ordering that exceeds or equals the actual upper bound, in case of such a vertex exists.

To update the point of regret, it is needed to analyze the model ν returned by the SAT solver, namely, the set of variables O, and its assignment. The routine that is responsible for update the point of regret performs a counting of variables $O_{i,j}$ that are set to true. If the number of variables $O_{i,j}$, exceeds or equals the actual upper bound, then the vertex i is designated as the new point of regret.

If it is impossible to find a new point of regret in the updated linear ordering, for the actual upper bound, then, the actual upper bound value is decremented. At each time that the actual upper bound value is decremented, the algorithm produces a verification to validate if the actual upper bound coincides with the lower bound, in that case, the upper bound value can not be decremented another time, hence, the algorithm outputs the treewidth value and stops. In the case of the lower and the upper bound values do not coincides, the point of regret is tested to the actual upper bound. In the case of a new point of regret is found for the actual upper bound value, the algorithm produces a new encoding, with the new linear ordering, the new upper bound, and the restored value c.

In the case of the *status* is set to UNSAT by the SAT solver, then, the value c is incremented with the *FREEVALUE*, in order to increment the size of the interval of the linear ordering in which the vertices are allowed to permute. Then, the SAT encoding is produced again, and the returned formula set will be analyzed by the SAT solver.

The local improv algorithm stops in two distinct cases. It stops in the case whether the actual upper bound is equal to the lower bound. And it stops in the case of the value of *status* is set to UNSAT by the SAT solver, and the vector of assumptions is empty. Note that in the case of the vector of assumptions is empty, the value c equals half the size of the linear ordering, which means that the left bound is set to the first position in the linear ordering, and the right bound is set to the last position in the linear ordering. Then, if the interval in which the vertices are allowed to permute, equals the size of the linear ordering, it means that all the vertices in the linear ordering are allowed to permute. Hence, if the SAT solver sets the *status* to UNSAT when all the vertices are allowed to permute in the encoding, we know that the actual upper bound fall behind the exact treewidth value. So, the last tested value for the treewidth is outputted, and the algorithm stops.

4. Results

In order to evaluate the solution that is described at section 3, it is provided a comparison between the implemented solution against successful existent ones: the one proposed by Gogate and Dechter[14] (QuickBB), and the winner of the PACE Challenge 2017 (Pace-Winner). Performing such an analysis it were used all the instances provided by the PACE challenge (the public and the private instances), the DIMACS Graph coloring networks instances[19], and the Named Graphs benchmarks instances[30].

The experiments were performed on a machine with an Intel(R) Xeon(R) CPU E5-2630 v2 proces-

sor (2.6GHz, 6C/12T), and 64 Gb of RAM, running Ubuntu 16.04. The algorithms are implemented in C++, and compiled with g++ version 5.4.0, having OpenWBO as the base architecture. To solve our SAT instances, we configured our software to run Glucose 4.1 as the default SAT solver. Running the algorithms, we limit the memory to 10 Gb for each process, and we enforce a CPU timeout at 3600s, for each instance.

To handle the time and memory restrictions, we used the *runsolver* application[27]. Such an application produce output files that allows one to monitor whether the time or memory restrictions were respected or not. Furthermore the application persists the output of each running instance in an output file, which can be later processed and analyzed.

4.1. Exact Treewidth

Comparing the implemented algorithms, we analyze how many instances were solved by each one of the algorithms, from each of the graph benchmarks described previously, at the beginning of this section. Moreover, it is analyzed the average time that each algorithm takes to solve the instances, in order to compare the average running time that a given algorithm spends to solve an arbitrary instance.

Additionaly, the implemented algorithms are compared against successful existent ones: the one proposed by Gogate and Dechter[14] (QuickBB), and the winner of the PACE Challenge 2017, which we call the Pace-Winner algorithm.

Tables 1, and 2 present the processed results, gathered by running each one of the algorithms over the set of instances. The rows of the tables represents the graph benchmarks used to run the algorithms, the columns represent the number of instances solved by each algorithm, as well as the average time that each algorithm spent to solve the instances.

The average size of the instances in the set of benchmarks, that is, the average number of vertices, and the average number of edges on each set of instances, is another property of the sets of benchmarks that is interesting to relate to the gathered results. Analyzing such a property, it is possible to establish a relation between the gathered results and the size of the graphs in the sets of benchmarks. The Pace-Challenge benchmarks have a number of average vertices and average edges considerably larger than the DIMACS coloring graph instances, and the named graphs instances. Thus, the SAT based algorithms solve a low number of instances in this set of benchmarks (13% of the instances). A possible explanation on the issue, is the number of variables produced to encode large graphs, since the logical model used to solve the treewidth problem produce $\mathcal{O}(n^3)$ variables, regarding the encoding of equation 1, which may overload the SAT solver decision process. However, analyze the impact of encoding a large number of variables, and the SAT formula structures, on the SAT solver decision process, is out of the scope of this work. For the named graphs instances, which presents the lowest number of vertices, and the lowest number of edges, in average, the SAT based algorithms solve 44% of the instances, which is a considerably larger number, than the number of instances solved in the Pace-Challenge benchmarks, in percentage.

Moreover, the average time that the algorithms spent to solve the instances in the named graphs benchmarks, is considerably lower than the average time that the algorithms taken to solve the instances in the Pace-Challenge benchmarks, and the average time that the algorithms spent to solve the instances in the coloring graphs benchmarks.

Observing the gathered results on the searching for the exact value for treewidth, it is possible to note that the performance of the algorithms varies considerably with the set of benchmarks used. Such a result can be observed by analyzing the number of instances solved by the Pace-Winner algorithm, for different benchmarks. The Pace-Winner algorithm clearly outperforms the other tested algorithms for the Pace-Challenge instances. However, when we go to the DIMACS Coloring Graphs benchmarks, the algorithm solves a considerable smaller number of instances than the branch-and-bound algorithm, and the SAT based algorithms proposed in the solution. Other interesting result that allows one to conclude that the performance of algorithms is closely related to the graphs topology, is the fact that the average time taken by the tested algorithm to solve the instances of named graphs benchmarks, is considerably inferior to the average time that the algorithm spent to solve the instances in the coloring graphs benchmarks, or in the Pace-Challenge set of instances.

Finally, we can conclude that the Pace-Winner algorithm outperforms the other tested algorithms, for the Pace Challenge and the named graphs benchmarks, and the QuickBB algorithm outperforms the other tested algorithms for the DIMACS coloring graphs benchmarks. Comparing only the SAT based algorithms for solving the treewidth value, the LinearSU-Inc algorithm outperforms the other algorithms, in the number of instances solved, and the LinearUS-Inc outperforms the other algorithms in the average time that is spent to solve each instance.

4.2. Treewidth Approximation

To find the best approximation algorithm we start by select all the instances that were not solved by all the tested algorithms, then we compare, for each instance, which was the algorithm that found the

	LinearSU		LinearUS		Binary		LinearSU-Inc		LinearUS-Inc		SelectedUS		LocalImprov	
	solved	time	solved	time	solved	time	solved	time	solved	time	solved	time	solved	time
Coloring	36	244.3	32	65.9	36	42.1	36	32.2	31	18.1	31	19.1	25	231
Named	65	3	66	30.9	65	2.9	66	35.1	65	7.7	65	9	65	4.7
PACE	23	114.2	24	104.6	26	92.3	22	142.4	22	53.7	21	66.4	22	59.8

Table 1: Instances solved and average time

	#Inst	QuickBB		Pace-Winner	
		solved	time	solved	time
Coloring	82	**43**	29.3	23	163
Named	150	81	46	**114**	27.1
PACE	200	23	56.7	**104**	856.4

Table 2: Instances solved and average time

	LinearSU	Binary	LinearSU-Inc	LocalImprov
Coloring	36	34	36	39
Named	27	28	30	72
PACE	110	95	107	146
Total	**173**	**157**	**173**	**257**

Table 3: Comparison on approximations

lowest value on the treewidth. Such a value represents the closer value to the exact value of the treewidth of the input graph, since it satisfies the SAT formula, and the algorithm keep searching for the next value, which means that the exact value was not found.

Comparing such values, every time that an algorithm found the lowest value for the treewidth, it receives one point. In the cases of multiple algorithms find the same best approximation, all that algorithms receives one point. In the end, the algorithm that received more points throughout the comparison process, it is meant to be the best approximation algorithm.

Table 3 shows the points attributed to each algorithm, for each one of the benchmarks tested and analyzed. The benchmarks are the same that we used to test the search SAT based method for find the exact value for the treewidth of a given input graph. The rows in the tables represents the benchmark instances used to evaluate the approximations of our solution, and the columns represents the algorithms that were tested and analyzed. Each one of the cells, represents the points that an algorithm achieved for each one of the benchmarks.

Analyzing the results presented in table 3, it is possible to observe that the LocalImprov algorithm outperforms any other algorithm as an approximation algorithm for treewidth, obtaining a total of 257 points, which corresponds to 87% of the instances tested. Furthermore, the algorithm achieves the optimal value for treewidth in the great part of the instances tested, running out of time to prove that the formula is unsatisfiable for the last tested value.

5. Conclusions

Treewidth is an NP-Hard problem which asks how close is a graph from being a tree. Solving such a problem have multiple applications in the real world, in computer science fields like artificial intelligence, bioinformatics, complex networks, or in combinatorics.

This document presents a SAT based approach to solve the treewidth value for a given input graph. Our solution consists on enabling pre-processing to bound the search space of SAT based algorithms, in order to accelerate the decisions process. Moreover, our solution aims at improving the previously existent logical model in the literature, as well as it consists on the implementation of specialized search algorithms to search for the treewidth value. Implementing such algorithms we designed new strategies, enabling incremental SAT solving in some of the algorithms, like LinearUS-Inc, LinearSU-Inc and SelectedUS. Furthermore, we implemented the LocalImprov algorithm, which operate by iteratively permute the vertices in the linear ordering given by the upper bound algorithm, until the permutation achieves an optimal linear ordering.

The implemented solution was tested on multiple well-known benchmarks, and was tested for find the exact value for the treewidth, and to achieve an approximation on the treewidth value for a given input graph. Performing the analysis of the results obtained, it is possible to conclude that LinearSU-Inc achieve the best results in the number of instances solved, and the LinearSU-Inc achieve the best results in the average time to compute an instance. Moreover, the LocalImprov algorithm was find to be the best algorithm for approximate the treewidth value of an arbitrary input graph.

For future work, we provide some ideas which can be explored in order to achieve better results on the searching for the treewidth value on graphs.

It would be interesting to design a framework that would pick an algorithm to search the treewidth value, respecting the class of a graph that is given as the input. This implies that a study must be performed to understand which algorithms performs the best for different classes of graphs. So, first of all, it would be necessary to study how some properties of graphs, as modularity and density, are related with the different existent algorithms to

solve the treewidth problem. Building these kind of relations allows one to construct a framework that, respecting the input graph properties, would pick an existent algorithm to solve the treewidth for that graph.

It is possible to turn exact methods into heuristic techniques. The main idea is to study how would perform our solution in the case we use it as an approximation heuristic. Such a technique would be based in setting a timeout for each running of the implemented solution, gathering which is the treewidth value at the time in which occurs the timeout, estimating the approximation for the exact value for the treewidth.

Acknowledgements

I would like to thank Professor Vasco Manquinho, who makes possible for me to work in such an interesting project, and learn more on graphs and algorithms, and for his patience and availability to share his knowledge throughout the process of development and writing this master thesis.

Moreover, I would like to thank to my family, who supports me all the time.

Finally, I would like to thank my friends and colleagues, with whom I have learned throughout these years.

References

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

[2] S. Arnborg and A. Proskurowski. Linear time algorithms for np-hard problems restricted to partial k-trees. *Discrete applied mathematics*, 23(1):11–24, 1989.

[3] J. Berg and M. Järvisalo. Sat-based approaches to treewidth computation: An evaluation. In *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, November 10-12, 2014*, pages 328–335. IEEE Computer Society, 2014.

[4] J. Berg, M. Järvisalo, and B. Malone. Learning optimal bounded treewidth bayesian networks via maximum satisfiability. In *Artificial Intelligence and Statistics*, pages 86–95, 2014.

[5] H. L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In *International Colloquium on Automata, Languages, and Programming*, pages 105–118. Springer, 1988.

[6] H. L. Bodlaender. Discovering treewidth. In P. Vojtás, M. Bieliková, B. Charron-Bost, and O. Sýkora, editors, *SOFSEM 2005: Theory and Practice of Computer Science, 31st Conference on Current Trends in Theory and Practice of Computer Science, Liptovský Ján, Slovakia, January 22-28, 2005, Proceedings*, volume 3381 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005.

[7] H. L. Bodlaender and A. M. Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 51(3):255–269, 2008.

[8] H. L. Bodlaender and A. M. Koster. Treewidth computations i. upper bounds., 2008.

[9] H. L. Bodlaender and A. M. C. A. Koster. Treewidth computations i. upper bounds. *Inf. Comput.*, 208(3):259–275, 2010.

[10] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[11] R. L. de Mantaras and L. Saitia. Quantified constraint satisfaction and bounded treewidth. In *ECAI 2004: 16th European Conference on Artificial Intelligence, August 22-27, 2004, Valencia, Spain: Including Prestigious Applicants [sic] of Intelligent Systems (PAIS 2004): Proceedings*, volume 110, page 161. IOS Press, 2004.

[12] H. Dell, T. Husfeldt, B. M. P. Jansen, P. Kaski, C. Komusiewicz, and F. A. Rosamond. The First Parameterized Algorithms and Computational Experiments Challenge. In J. Guo and D. Hermelin, editors, *11th International Symposium on Parameterized and Exact Computation (IPEC 2016)*, volume 63 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:9, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[13] G. Elidan and S. Gould. Learning bounded treewidth bayesian networks. *Journal of Machine Learning Research*, 9(Dec):2699–2731, 2008.

[14] V. Gogate and R. Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 201–208. AUAI Press, 2004.

[15] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural csp decomposition methods. *Artificial Intelligence*, 124(2):243–282, 2000.

[16] J. Gramm, A. Nickelsen, and T. Tantau. Fixed-parameter algorithms in phylogenetics. *The Computer Journal*, 51(1):79–101, 2007.

[17] M. T. Hajiaghayi. *Algorithms for graphs of (locally) bounded treewidth*. PhD thesis, University of Waterloo, 2001.

[18] X. Huang and J. Lai. Parameterized graph problems in computational biology. In *Computer and Computational Sciences, 2007. IMSCCS 2007. Second International Multi-Symposiums on*, pages 129–132. IEEE, 2007.

[19] D. S. Johnson and M. A. Trick. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, volume 26. American Mathematical Soc., 1996.

[20] J. Korhonen and P. Parviainen. Exact learning of bounded tree-width bayesian networks. In *Artificial Intelligence and Statistics*, pages 370–378, 2013.

[21] A. M. Koster, S. P. van Hoesel, and A. W. Kolen. Solving partial constraint satisfaction problems with tree decomposition. *Networks: An International Journal*, 40(3):170–180, 2002.

[22] J. Lagergren. Efficient parallel algorithms for graphs of bounded tree-width. *Journal of Algorithms*, 20(1):20–44, 1996.

[23] D. Lokshtanov, D. Marx, and S. Saurabh. Known algorithms on graphs of bounded treewidth are probably optimal. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*, pages 777–789. Society for Industrial and Applied Mathematics, 2011.

[24] R. Martins, V. Manquinho, and I. Lynce. Open-wbo: A modular maxsat solver. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 438–445. Springer, 2014.

[25] P. Parviainen, H. S. Farahani, and J. Lagergren. Learning bounded tree-width bayesian networks using integer linear programming. In *Artificial Intelligence and Statistics*, pages 751–759, 2014.

[26] N. Robertson and P. D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.

[27] O. Roussel. Controlling a solver execution with the runsolver tool system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:139–144, 2011.

[28] M. Samer and H. Veith. Encoding treewidth into sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 45–50. Springer, 2009.

[29] M. Samer and H. Veith. Encoding treewidth into SAT. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 45–50. Springer, 2009.

[30] E. Weisstein. Mathworld online mathematics resource (2016). *Last accessed on February*, 13:2015, 2015.

[31] J. Zhao, D. Che, and L. Cai. Comparative pathway annotation with protein-dna interaction and operon information via graph tree decomposition. In *Biocomputing 2007*, pages 496–507. World Scientific, 2007.

[32] J. Zhao, R. L. Malmberg, and L. Cai. Rapid ab initio prediction of rna pseudoknots via graph tree decomposition. *Journal of mathematical biology*, 56(1-2):145–159, 2008.