Treewidth applications for Combinatorial Optimisation Problems

Seminar: Algorithms on Sparse Graphs

Luca Oeljeklaus
luca.oeljeklaus@rwth-aachen.de

August 22, 2018

In this paper we will examine the applicability of the concept of treewidth in solving usually NP*-hard problems efficiently. We will specifically give an overview of algorithms capable of solving the classical Maximum Independent Set Problem efficiently on specific graph classes.*

Contents

1 Introduction

In this paper, we provide an overview of the uses of the notion of treewidth for solving combinatorial optimisation problems. More precisely, we will study algorithms that solve the Maximum Independent Set Problem (MIS) efficiently when provided with information about the treewidth of a graph.

First, we will be introducing the notions of tree decomposition and treewidth. Then we will proceed by considering algorithms solving the MIS problem on trees, SP-graphs and graphs of bounded treewidth, finishing with a polynomial-time approximation scheme for the MIS problem on planar graphs.

All introduced algorithms work on undirected graphs, given by $G = (V, E)$ with V a set of vertices and $E \subseteq (V \times V)$ a set of edges. We call a subgraph G' of G induced by $W \subseteq V$ if $G' = (W, E \cap (W \times W))$.

2 Introducing Treewidth

2.1 Tree decomposition

The concept of *tree decomposition* has been separately introduced by Halin in 1976 [1] and by

Robertson and Seymour in 1984 [2]. Intuitively, given an arbitrary graph, its tree decomposition will determine which vertices must be merged to obtain a tree. Such a tree decomposition is not unique. Examples for a graph and a corresponding tree decomposition can be found in Figure 1 and Figure 2.

Definition 1

Given a graph $G = (V, E)$, a *tree decomposition* of G is given by a pair $(\{ X_i \mid i \in I \}, T = (I, F))$ where T is a tree and each node[1] $i \in I$ is associated to a subset of vertices (referred to as bags) $X_i \subseteq V$ such that:

- $\bigcup_{i \in I} X_i = V$.
  Each vertex $v \in V$ is contained by at least one bag X_i.
- $\forall (v, w) \in E, \exists i \in I : v, w \in X_i$.
  For any edge, there is a bag X_i containing both its endpoints.
- $\forall v \in V : \{ i \in I \mid v \in X_i \}$ induces a subtree of T.
  All the nodes whose bags contain the same vertex $v \in V$ are connected and do not form a cycle.

The *width* of a tree decomposition $(\{ X_i \mid i \in I \}, T = (I, F))$ is given by $\max_{i \in I} |X_i| - 1$, the size of its biggest bag minus one.

2.2 Treewidth

From this definition of tree decomposition we can derive the definition of *treewidth*.

Definition 2

Given a graph $G = (V, E)$, its treewidth is given by the minimum width over all its tree decompositions.

Given all tree decompositions of a graph, the treewidth is defined as the minimum width over all of its decompositions. The -1 is used for aesthetic reasons as it lets trees have treewidth 1.

[1] We refer to vertices of tree decompositions as nodes.

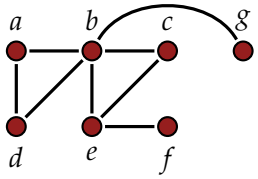

Figure 1: Graph G_α.

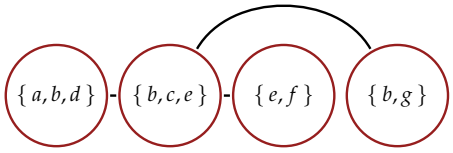

Figure 2: A tree decomposition for G_α.

2.3 Treewidth approximation

In general, the problem of determining if a graph has a treewidth smaller than k for a given k is NP-complete. However, it is possible to approximate treewidth. The currently best known polynomial-time approximation algorithm provides a $\sqrt{\log k}$-approximation [3].

Thus, discovering if a tree decomposition of width k exists without knowing the treewidth is also NP-complete. However, for constant k, it is possible to find such a tree decomposition in linear time [4]. This means that the treewidth problem is in FPT.

3 MIS on Trees

3.1 Treewidth of trees

The class of trees is equal to the class of graphs with treewidth 1. This is because the presence of a cycle forces at least 3 nodes to be in the same bag, therefore any graph with a cycle has a treewidth of at least 2. Conversely, if a graph is a tree, there always exists a tree decomposition such that each node contains at most 2 vertices.

3.2 Algorithm

Determining the size of the MIS for trees can be achieved in linear time using an algorithm described by Bodlaender [5]. This is done by choos-

ing an arbitrary vertex as a root and then recursively computing two values A and B for each vertex v:

- A: the size of the MIS of the subtree induced by v including v.
- B: the size of the MIS of the subtree induced by v excluding v.

These values are given for leaves as $(A, B) = (1, 0)$ as the subtree induced by a leaf is only the leaf itself, so either it is itself contained in the independent set or not.

For a non-leaf node v with children $C = \{ c_1 \dots c_n \}$, the values are given by:

$$(A, B) = \left(\sum_{i=1}^{n} c_i(B), \sum_{i=1}^{n} \max \left(c_i(A), c_i(B) \right) \right).$$

That is, the A value is the sum of B values of the children. This is due to the fact that if a vertex v lies in the set, none of its direct children can, as this would result in a violation of the independence property. The B value on the other hand allows for children to either be inside or outside of the independent set.

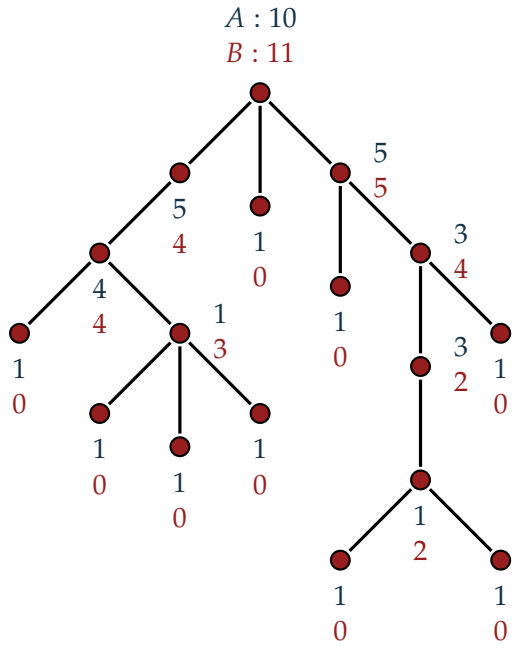

Figure 3: MIS sizes computed for all nodes of a tree.

An example of this algorithm can be seen in Figure 3. The final result of the algorithm is the maximum of the A and B values of the root. In Figure 3, the size of the MIS is thus given by $B = 11$.

4 MIS on SP-graphs

4.1 Treewidth of SP-graphs

The class of *series-parallel graphs* or SP-graphs is a subset of the graphs of treewidth 2 [6].

4.2 Defining SP-graphs

SP-graphs are graphs that are defined by a tuple $(G = (V, E), s, t)$, with a graph G, a source s and a sink t. Such a graph must be definable by recursive application of two operations: *serialisation* and *parallelisation*, the base case being two vertices and an edge connecting them. An important restriction is that for an SP-graph, source and sink are fixed and may not be changed. For example, (G_β, d, f) and (G_β, f, d) are two different graphs, and (G_β, b, f) is not a valid SP-graph as it cannot be constructed.

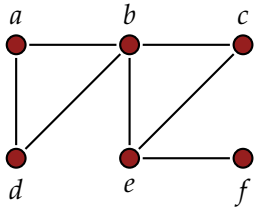

Figure 4: An SP-graph (G_β, d, f).

Parallelisation Given two SP-Graphs (G_x, s_1, t_1) and (G_y, s_2, t_2), they can be parallelised by identifying s_1 and s_2 as well as t_1 and t_2 (see Figure 5).

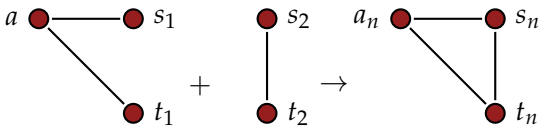

Figure 5: Parallelisation of two SP-Graphs.

Serialisation Given two SP-Graphs (G_x, s_1, t_1) and (G_y, s_2, t_2), they can be serialised by identifying t_1 and s_2 into a single vertex (see Figure 6).

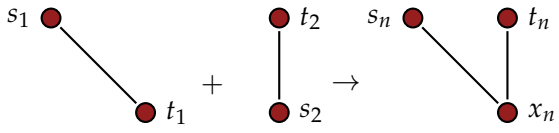

Figure 6: Serialisation of two SP-Graphs.

4.3 SP-trees

For such an SP-graph, it is possible to build an *SP-tree*. In this tree structure, there are two types of binary and one type of nullary vertices, all representing subgraphs of the original graph and each labelled with the respective source and sink:

- P nodes correspond to the parallelisation of their two children.
- S nodes correspond to the serialisation of their two children.
- Leaves correspond to edges of the original graph.

For example, the subgraph of G_β induced by the left subtree of the root of Figure 7 is the graph induced by the vertices $\{ d, a, b \}$ as we first serialise da and ab and then parallelise the result with db.

Such an SP-tree can be computed efficiently [7] and is not unique.

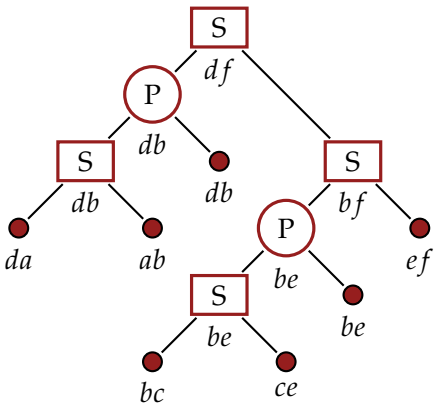

Figure 7: An SP-tree for (G_β, d, f).

4.4 Algorithm

On this SP-tree structure, we can now run a recursive algorithm [5] similar to the one in section 3.2 to compute the MIS. However, instead of computing two values, we will be computing four values for each vertex:

- AA: the size of the MIS containing both s and t.
- AB: the size of the MIS containing s but not t.
- BA: the size of the MIS not containing s but t.
- BB: the size of the MIS containing neither s nor t.

These values are trivial for leaves (i. e. edges of the original graph), as they are given by $(AA, AB, BA, BB) = (-\infty, 1, 1, 0)$. For S and P nodes with children l and r, they are given by the following rules (where l and r are the left and right children respectively):

P node:

- $AA = AA(l) + AA(r) - 2$
- $AB = AB(l) + AB(r) - 1$
- $BA = BA(l) + BA(r) - 1$
- $BB = BB(l) + BB(r)$

S node:

- $AA = \max\{AA(l) + AA(r) - 1, AB(l) + BA(r)\}$
- $AB = \max\{AA(l) + AB(r) - 1, AB(l) + BB(r)\}$
- $BA = \max\{BA(l) + AA(r) - 1, BB(l) + BA(r)\}$
- $BB = \max\{BA(l) + AB(r) - 1, BB(l) + BB(r)\}$

For P nodes, we add the values of the children and subtract 0, 1 or 2 to avoid double-counting the source or the sink. For S nodes, the question we ask is if we can keep the merged vertex without breaking the independence property. The size of the MIS is the maximum over the four values of the root, i. e. 3 in Figure 8.

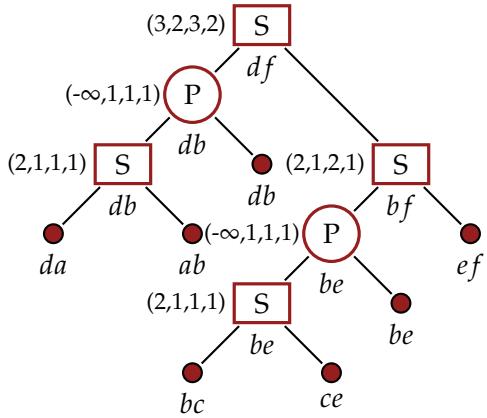

Figure 8: MIS calculations for (G_β, d, f).

5 MIS on graphs of small treewidth

5.1 Nice tree decomposition

The algorithm we will be introducing is based on a more restrictive version of the tree decomposition called a nice tree decomposition.

Definition 3

A *nice tree decomposition* has all the properties of a tree decomposition. Additionally, each node is of exactly one of the following types:

- **Leaf Node**: i is a leaf of T and $|X_i| = 1$.
- **Join Node**: i has two children j_1 and j_2 and $X_i = X_{j_1} = X_{j_2}$.
- **Forget Node**: i has one child j and $\exists v \in V$ s.t. $X_i = X_j \setminus \{ v \}$.
- **Introduce Node**: i has one child j and $\exists v \in V$ s.t. $X_i = X_j \cup \{ v \}$.

Given a non-nice tree decomposition of width k, a nice one of the same width can be discovered in linear time [8] using the following algorithm:

Algorithm 1

Input: A tree decomposition of width k.
Output: A nice tree decomposition of width k.

1. Choose an arbitrary node as root.
2. If a node v_1 has more than 2 children $\{ c_1, \ldots, c_n \}$:
   a) Create copy v_2 of v_1.
   b) Add v_2 as a child of v_1.
   c) Set $\{ c_2, \ldots, c_n \}$ as children of v_2.
   d) Repeat until reaching v_{n-1}, which will have children $\{ c_{n-1}, c_n \}$.
3. For nodes v with 2 children $\{ c_1, c_2 \}$:
   a) Create copies of v: v_1 and v_2.
   b) Set $\{ v_1, v_2 \}$ as children of v.
   c) Set c_x as child of v_x.
4. For nodes with 1 child, create a series of introduce and forget nodes between them and their child.
5. For leaf nodes, create a series of introduce nodes so that the leaf has size 1.

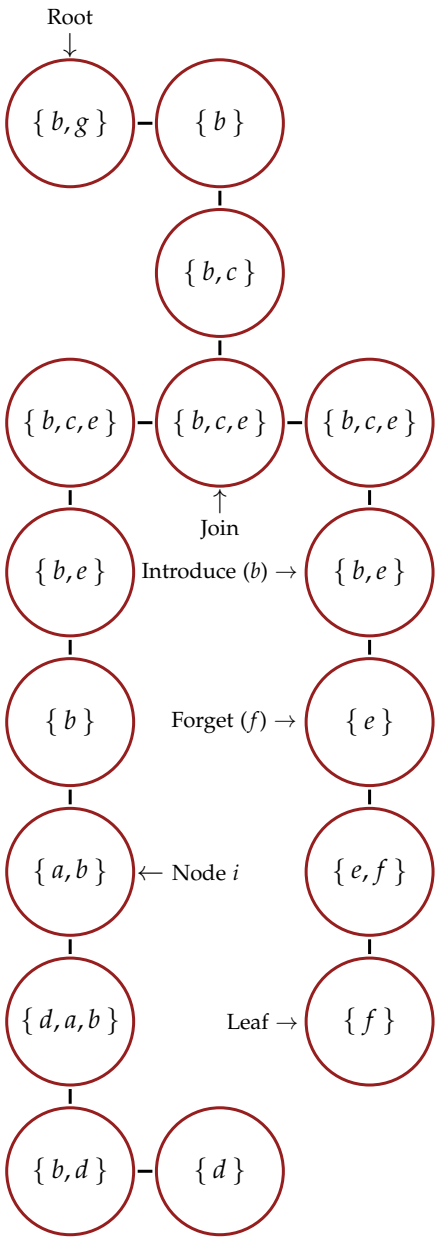

Figure 9: Nice tree decomposition of G_α.

In Figure 9 we can see the result that Algorithm 1 provides when given the tree decomposition from Figure 2. The diagram also includes labelling to simplify the understanding of the 4 types of nodes.

5.2 Algorithm

For each node i with descendants $\{ d_1, \ldots, d_m \}$, we define the graph $G_i = (V_i, E_i)$ such that $V_i = \bigcup_{y \in \{1,\ldots,m\}} X_{d_y}$ and $E_i = E \cap (V_i \times V_i)$. That is, G_i is the subgraph of G induced by all the vertices contained by i and its descendents.

Further, we will compute a table C_i for each node i. These tables will contain an entry for each subset $S \subseteq X_i$. The rules for computing them will be given first in Algorithm 2 and then further explained below.

As an example, the table on the the next page provides C_i for the node i from Figure 9.

$C_i(\varnothing)$	0
$C_i(\{ a \})$	1
$C_i(\{ b \})$	1
$C_i(\{ a, b \})$	$-\infty$

Table 1: C_i for node i in Figure 9.

The entries $C_i(S)$ represent the maximum sizes of the independent set in G_i containing the vertices in S. If some vertices in S do not fulfil the independence criterion (meaning that they are neighbours), the corresponding entry will be $-\infty$.

Algorithm 2

Input: A nice tree decomposition $(\{ X_i \mid i \in I \}, T = (I, F))$ of width k for a graph G.
Output: The size of the MIS for G.

1. Compute a postorder (LRN) walk for T.
2. In the order of the LRN walk, compute the table C_i for each node i:
   - For leaf nodes i with $X_i = \{ v \}$:
     - $C_i(\varnothing) = 0$ and $C_i(\{ v \}) = 1$.
   - For join nodes i with children j_1, j_2 and $X_i = X_{j_1} = X_{j_2}$ and $S \subseteq X_i$:
     - $C_i(S) = C_{j_1}(S) + C_{j_2}(S) - |S|$.
   - For forget nodes i with child j and $X_i = X_j \setminus \{ v \}$ and $S \subseteq X_i$:
     - $C_i(S) = \max \{ C_j(S), C_j(S \cup \{ v \}) \}$
   - For introduce nodes i with child j and $X_i = X_j \cup \{ v \}$ and $S \subseteq X_j$:
     - $C_i(S) = C_j(S)$
     - $C_i(S \cup \{ v \}) = \begin{cases} -\infty, & \text{if } \exists w \in S : \{ v, w \} \in E, \\ C_j(S) + 1, & \text{else.} \end{cases}$
3. Return the largest value from C_r, the table for the root r.

The computation of C_i for leaf nodes is trivial, as the induced subgraph is by definition the single vertex contained in the bag X_i. Thus either the vertex itself lies in the independent set, or it doesn't.

For a join node i, both childrens bags X_{j_1} and X_{j_2} contain by definition the same vertices. Because of the third property of tree decompositions, any node contained in both subtrees must also be contained in X_i, so we can make the assertion that the nodes contained in X_i are exactly the nodes that both subgraphs G_{j_1} and G_{j_2} have in common. So we can, for any $S \subseteq X_i$, sum over X_{j_1} and X_{j_2} and subtract those that we counted twice. In Figure 9, we can see that the subgraphs of the subtrees of the join node induce those given by the vertices $\{ b, c, e, a, d \}$ and $\{ b, c, e, f \}$ in G_α respectively, and that the intersection of both is exactly $\{ b, c, e \}$.

For a forget node i with child j and forgotten vertex v, what we do is check if for any $S \subseteq X_i$, the set $S \cup \{ v \}$ preserves the independence property. If it does, we keep it, else we reject it because it would have size $-\infty$.

For an introduce node i with child j and added vertex v, the first thing we do is take over the values for all $C_i(S)$ with $S \subseteq X_j$. For those sets with $v \in S'$, we test if any vertex within S' has an edge connecting it to v. If so, this breaks the independence property and we return $-\infty$, else we add 1 for the added vertex (note that this might still be $-\infty$ if S' itself is not independent).

This algorithm was introduced by Bodlaender [5].

5.3 Complexity

Following the postorder walk has complexity $\mathcal{O}(n)$. As the largest bag contains $k+1$ elements (assuming treewidth k), we need to compute at most 2^{k+1} entries for each table C_i. Thus, we obtain a complexity of $\mathcal{O}(2^{k+1}n)$ for this algorithm.

This means that for graphs with bounded treewidth, we have a linear time algorithm for computing the maximum independent set, as computing a tree decomposition [4], transforming it into a nice tree decomposition and finding a post order walk all have complexity $\mathcal{O}(n)$.

Let it be noted, that, while technically efficient as long as the treewidth is bounded, the 2^{k+1} in the complexity gets very large nonetheless once the treewidth increases. This makes it in practice only useful for graphs of "small" treewidth.

6 A PTAS for MIS on planar graphs

6.1 Remarks and definitions

A graph is called *planar* if it can be embedded into the plane such that there are no intersecting edges. The problem of solving the MIS problem on planar graphs is known to be NP-hard [9].

A graph is called (1)-*outerplanar* if there exists an embedding such that every vertex lies on the outer face of the graph. A graph is called k-*outerplanar* if removing all outer vertices (and incident edges) results in a $(k-1)$-outerplanar graph. The treewidth of a k-outerplanar graph is bounded by $3k-1$ [10].

For an arbitrary t-outerplanar graph, the layers of vertices are referred to by the sets L_1,L_2,...,L_t, with L_1 the outermost and L_t the innermost layer.

We will use the aforementioned treewidth bound for k-outerplanar graphs in a *polynomial-time approximation scheme*. Such a *PTAS* has a polynomial complexity and provides approximate results for the problem it is given. More precisely, for minimisation problems, a PTAS provides a $(1+\varepsilon)$-approximation for an $\varepsilon > 0$, and a $(1-\varepsilon)$-approximation for minimisation problems.

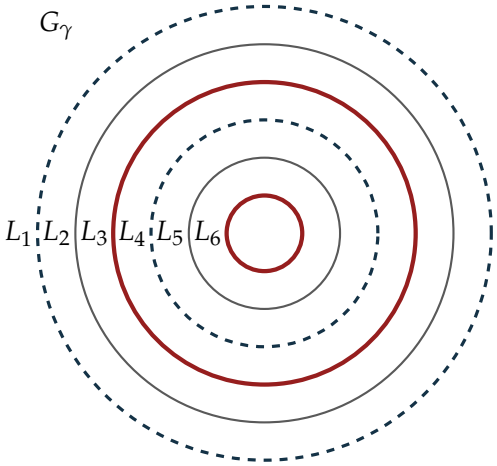

Figure 10: Example for G_γ with 6 layers.

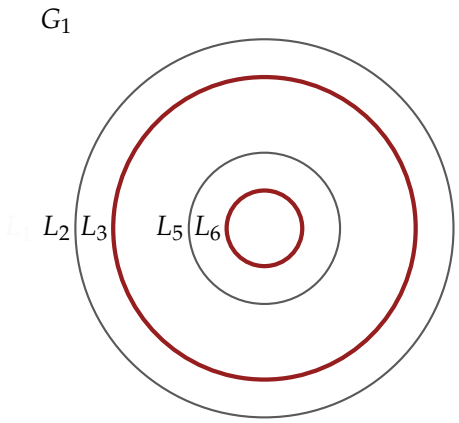

Figure 11: G_1 for G_γ with $\varepsilon = \frac{1}{3}$ and thus $l = 3$, L_1 and $L_{1+3=4}$ removed.

6.2 Algorithm

We now introduce a PTAS that will, for any planar graph, provide an $(1-\varepsilon)$-approximation of the maximum independent set in polynomial time [5].

Assume that we are given a planar graph G_γ with layers L_1,...,L_t. First, we choose an $\varepsilon > 0$ (which will typically be some fraction smaller than 1) and set $l = \lceil \frac{1}{\varepsilon} \rceil$. We then define $G_i = (V_i, E_i)$ for $i \in \{1, \ldots, l\}$ such that:

- $V_i = V_\gamma \setminus \{ L_{i+0k}, L_{i+1k}, L_{i+2k}, \ldots \}$.
  We remove every l-th layer starting at the i-th.
- $E_i = E_\gamma \setminus \{ \{v, w\} \mid v \notin V_i \vee w \notin V_i \}$.
  We remove all edges of which at least one endpoint is not in V_i.

By construction, each of these G_i is at most $(l-1)$-outerplanar. Thus the treewidth for each G_i is bounded by $3l-1$, as has been stated in section 6.1. Since we have an upper bound on the treewidth, we can compute a tree decomposition of width $3l-1$ in polynomial time [10] for each G_i. By applying Algorithm 1 we can convert this into a nice tree decomposition in linear time. We can then compute the MIS for each G_i in polynomial time using Algorithm 2.

As there are l G_i's and the computation of the MIS for each has a complexity of $\mathcal{O}(2^{3l-1} \cdot n)$, this PTAS has a complexity of $\mathcal{O}(2^{3l-1} \cdot l \cdot n)$.

After computing the MIS for each G_i, we return the one with the highest cardinality.

6.3 Proof of goodness

To prove that the independent set that this PTAS provides really is an $(1-\varepsilon)$-approximation, we assume W to be a maximum independent set. Then we can define $W_i = W \cap V_i$, that is, the vertices in the MIS that lie in the graph G_i. By construction it holds that:

$$\begin{aligned} & \forall v \in W. \exists! i \in \{1, \ldots, l\} : v \notin W_i \\ \Rightarrow \quad & \textstyle\sum_{i=1}^{l} |W_i| = (l-1) \cdot |W| \\ \Rightarrow \quad & \exists i \in \{1, \ldots, l\} : \end{aligned}$$

$$\begin{aligned} |W_i| &\geq \tfrac{1}{l} \textstyle\sum_{i=1}^{l} |W_i| \\ &\geq \tfrac{1}{l} \cdot (l-1) \cdot |W| \\ &\geq \tfrac{1}{\frac{1}{\varepsilon}} \cdot (\tfrac{1}{\varepsilon} - 1) \cdot |W| \\ &\geq \tfrac{\frac{1}{\varepsilon} - 1}{\frac{1}{\varepsilon}} \cdot |W| \\ &\geq \varepsilon \cdot \left(\tfrac{1}{\varepsilon} - 1\right) \cdot |W| \\ |W_i| &\geq (1-\varepsilon) \cdot |W| \end{aligned}$$

Plainly this means that for each vertex in W on layer i, there is exactly one G_x that does not contain it, namely the one that removes layer i. Thus, the sum over the elements of all W_i is the same as taking l times W but removing each layer once, thus obtaining $(l-1) \cdot |W|$.

Since there must be at least one W_i larger or equal than the average W_j, we can simply set up that inequation and transform the equation until we find that $|W_i| \geq (1-\varepsilon) \cdot |W|$, which is exactly what we were trying to show.

7 Conclusion

After introducing the notions of tree decomposition and treewidth along with other concepts, we have looked at their applications on usually NP-complete problems. We have seen algorithms solving the maximum independent set problem efficiently on trees, SP-graphs and graphs of small treewidth. The last algorithm we introduced was a polynomial-time approximation scheme capable of approximating the maximum independent set of arbitrary planar graphs.

This means that, for applications which can make assumptions about the properties of the graphs they work on, algorithms that exploit treewidth and related concepts can be powerful tools in reducing the complexity of usually hard problems. If an application only requires an approximate solution, this in general represents a computational advantage. This is, as we have seen, also the case for algorithms exploiting the treewidth of graphs.

References

[1] Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8:171–186, 1976.

[2] Neil Robertson and Paul D. Seymour. Graph minors iii: Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[3] Uriel Feige, MohammadTaghi Hajiaghayi, and James R. Lee. Improved approximation algorithms for minimum-weight vertex separators. *Proceedings of the 37th Annual Symposium on the Theory of Computing*, pages 563–572, 2005.

[4] Hans L. Bodlaender. A linear-time algorithm for finding tree decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996.

[5] Hans L. Bodlaender and Arie M. C. A. Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 51(3):255–269, 2007.

[6] Hans L. Bodlaender and Babette van Antwerpen-de Fluiter. Parallel algorithms for series parallel graphs and graphs with treewidth two. *Algorithmica*, 29:543–559, 2001.

[7] David Eppstein. Parallel recognition of series-parallel graphs. *Information and Computation*, 98(1):41–55, 1992.

[8] Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer, 1994.

[9] Vladimir E. Alekseev, Vadim Lozin, Dmitriy Malyshev, and Martin Milanič. *The Maximum Independent Set Problem in Planar Graphs*, volume 5162 of *Lecture Notes in Computer Science*. Springer, 2008.

[10] Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209(1):1–45, 1998.