Bangladesh University of Engineering and Technology

Department of Computer Science and Engineering

Academic Year 2022 - 2023

CSE 462

-Algorithm Engineering Sessional-

Report On

PACE 2017 – Track A: Tree Width

Group Members:

1. 1805006 - Tanjeem Azwad Zaman
2. 1805008 - Abdur Rafi
3. 1805010 - Anwarul Bashir Shuaib
4. 1805019 - Md Rownak Zahan Ratul
5. 1805030 - Md Toki Tahmid

Section: A1

Date of Submission: March 09, 2024

Contents

1 Problem Definition

Our project was inspired by the PACE (Parameterized Algorithms and Computational Experiments) challenge, specifically the problem focused in the 2016 and 2017[1] version of the challenge : The Treewidth Problem
In the realm of algorithmic graph theory, the concepts of treewidth and tree decomposition are pivotal constructs for addressing computational problems. The treewidth of a graph effectively quantifies its resemblance to a tree, serving as a crucial parameter that can drastically influence the computational complexity of various graph problems. A graph with a low treewidth suggests a tree-like structure, which often allows for more efficient algorithmic solutions that would be infeasible on graphs with higher treewidth. Tree decompositions, on the other hand, provide a methodical framework to exploit this tree-like architecture, by partitioning the graph into interconnected subsets (called 'bags') that conform to specific properties. These properties ensure that each graph cycle is "caught" in the structure of the tree decomposition, thereby enabling dynamic programming techniques to traverse and process the graph in a systematic and computationally viable manner. The study and application of treewidth and tree decomposition algorithms have profound implications in algorithm engineering, paving the way for the development of more effective algorithms across a multitude of graph-based applications.

1.1 Tree Decomposition

Finding the tree decomposition with the minimal treewidth enables efficient and systematic dynamic programming algorithms. And finding such a tree decomposition often goes hand-in-hand with finding the treewidth, which is an important parameter for a graph in many problems.

1.1.1 Introduction

Given a graph G, with Vertex set V and edge set E,

- A ***tree decomposition*** of G is a ***tree-like structural representation*** of G
- Each node of the decomposition corresponds to a ***Bag***
- Each ***bag*** is a subset of the original vertex set V
- A graph can have multiple tree valid tree decompositions

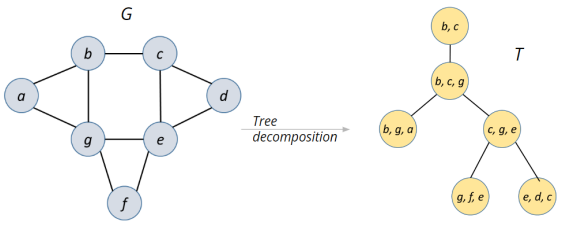
Furthermore, a tree decomposition must follow 3 conditions to be considered valid. Details are described herewith.

1.1.2 Definition

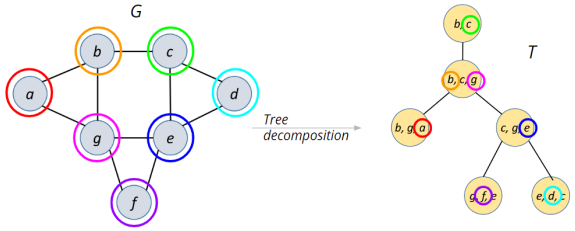
A tree decomposition is represented as: $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$, where

Formal	**Informal**
T is a tree representing the structure of the decomposition	The first element represents a Tree-like structure
$\{X_t\}\ \forall t \in V(T), X_t \subseteq V(G)$	Each bag (corresponding to a tree node) is a subset of V(G). The second element is the set of all such bags.
And the following 3 properties hold:	
1. $\bigcup_{t \in V(T)} X_t = V(G)$	Every vertex of G is in at least 1 bag of T
2. $\forall (u, v) \in E(G), \exists$ a node t in T, s.t both u and v belong to X_t	For all edges in G, there is at least 1 bag in T that has both endpoints of the edge
3. $\forall u \in V(G)$, the set $T_u = \{t \in V(T) : u \in X_t\}$	All bags that contain any specific vertex of G, make a connected subtree in T

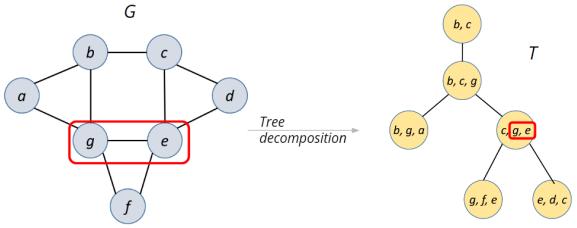
1.1.3 An Example:



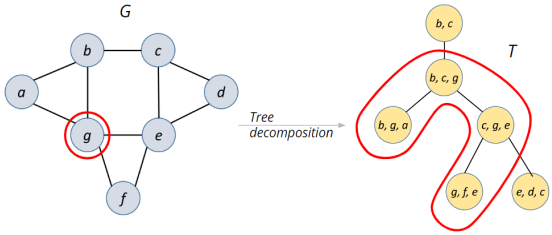
(a) A Tree Decomposition T of the graph G



(b) Each node in G belong to a bag of T *[Condition 1]*



(c) For all edges, at least 1 bag has both endpoints. Edge ge is an example *[Condition 2]*



(d) All bags with a vertex (g) form a connected component *[Condition 3]*

Figure 1: Example of a Tree Decomposition

1.2 Treewidth

Treewidths and their associated tree decompositions play a pivotal role within the domain of fixed-parameter tractable (FPT) algorithms and parameterized complexity. In the realm of parameterized algorithms, one aims to isolate certain parameters — independent of the input size — to devise more efficient computational strategies. Treewidth is often employed as such a parameter, primarily because many graph problems, deemed intractable in general scenarios, become manageable when the treewidth is bounded; allowing for the efficient solving of problems such as Hamiltonian path, graph coloring, and maximum independent set when the treewidth is small.

1.2.1 Definitions:

- ***Width of a Bag:*** Size of the bag - 1
- ***Width of a Tree Decomposition:*** Maximum width among all bags in the Decomposition
- ***Treewidth of a graph:*** Minimum width among all tree decompositions of the graph

1.2.2 Example:

Looking at figure 1a, we see that the bags are $\{b, c\},\ \{b, g, a\},\ \{b, c, g\},\ \{c, g, e\},\ \{g, f, e\},\ \{e, d, c\}$ with widths 1, 2, 2, 2, 2, 2. Thus the width of this tree decomposition is max(1,2,2,2,2,2) = 2. Coincidentally, this is the lowest among all tree decompositions, and this decomposition is valid with a tree width of 2.

1.3 Formal Problem Definition

1.3.1 Optimization Version:

Given a graph G, find its treewidth (and a tree decomposition that gives this treewidth)

1.3.2 Decision Version:

Given a graph G and an integer k; is the treewidth of G at most k?

2 List of Existing Algorithms

2.1 Exact Algorithms

Algorithm Name	Runtime/ Approx Ratio	Authors	Year	Comments
Linear-Time Algorithm for small treewidth [2]	$2^{O(k^3)} \cdot O(n)$	Hans L. Bodlaender	1996	• algorithm for small treewidth in linear time • treewidth is fixed.
PID - ACP (DP based Algorithm) [3]	$O(n^{\frac{k+2}{2}})$	Hisao Tamaki	2016	• 1st in PACE 2016: Exact Track
PID - Bouchitte Todinca [4]	$O(I\text{-block}^k * O\text{-block}^k)$	Hisao Tamaki	2017	• 2nd in PACE 2017: Exact Track • Based on minimal separators and potential maximal cliques
Jdrasil: A Modular Library for Computing Tree Decompositions [5]	Library of several algorithms	Max Bannach, Sebastian Berndt, and Thorsten Ehlers	2017	• 3rd in PACE 2017: Exact Track • Supports parallel processing
Large induced subgraphs via triangulations and CMSO [6]	$O(1.7347^n)$	Fedor Fomin, Ioan Todinca, Yngve Villanger	2015	• Based on minimum triangulation

Table 1: Exact Algortihms

2.2 Approximation Algorithms

Algorithm Name	Runtime/ Approx Ratio	Authors	Year	Comments
Finding all leftmost separators of size [7] $\leq k$	Runs in $2^{6.75k}$, Approximation Ratio: $O(n \log n)$	Belbasi & Fürer	2021	Focuses on improving the exponential value related to "k"
An Improved Parameterized Algorithm for Treewidth [8]	Runs in $2^{O(k^2)}n^{O(1)}$, Approximation ratio: $(1+\varepsilon)k \quad [\varepsilon \in (0,1)]$	Tuukka Korhonen, Daniel Lokshtanov	2022	First improvement on the dependency on k in algorithms for treewidth since the $2^{O(k^3)}n^{O(1)}$ time algorithm given by Bodlaender and Kloks [ICALP 1991]

Table 2: Approximation Algorithms

2.3 Heuristic and Metaheuristic Algorithms

Algorithm Name	Authors	Year	Comment
FlowCutter PACE17 [9]	Hamann, M. & Strasser, B.	2017	Heuristic
TreewidthDP (Dynamic Programming) [10]	Koster, R., Bodlaender, H.L., & Van Hoesel, S.P. (2005).	2005	Exact / Heuristic
Genetic Algorithm for Treewidth [11]	Gogate, V. & Dechter, R.	2002	Metaheuristic
Arnborg & Proskurowski's Heuristic [12]	Arnborg, S. & Proskurowski, A. (1989)	1989	Heuristic

Table 3: Algorithms for Treewidth

3 NP-Completeness of Treewidth

The NP-completeness of treewidth computation was proved in 1987 by Arnborg et al. [13]. The reduction of the problem involves the Minimum Linear Cut Arrangement (MCLA) problem. The following sections provide the required definitions for the reduction.

3.1 Definitions

3.1.1 k-chordal graphs

Given $G(V, E)$, G is k-Chordal if,

1. G is a chordal graph.
2. G has a maximum clique size of $k + 1$.

Facts:

- Filled Graph: Graph $G(V, E \cup F')$ after adding all fill edges F' with respect to an elimination scheme.
- A k-tree with more than k vertices is k-chordal.

3.1.2 Block-contiguous elimination scheme

A block is defined as a set of maximal vertices that share the same neighbourhood. An elimination scheme is said to be block-contiguous if the vertices that form a block are ordered contiguously.

Facts:

- There always exists a block-contiguous elimination order for the corresponding filled graph of a minimal chordal supergraph of a bipartite graph.

3.1.3 Minimum Cut Linear Arrangement Problem

Given a graph $G(V, E)$ and a positive integer k, the question is whether there exists a permutation π of V such that the linear cut value $c_\pi(G) \leq k$, i.e., is the linear cut value of G less than or equal to k?

The linear cut value, $c_\pi(a)$, for a given arrangement a is determined by the maximum number of edges crossing any cut in the linear arrangement, formally defined as:

$$c_\pi(a) = \max_{1 \leq i < |V|} |\{(u, v) \in E \mid r(u) < i < r(v)\}|$$

where r represents the rank function assigning each vertex to its position in the permutation π.

3.2 Reduction: MCLA $\leq_p$ Treewidth

K-trees can be recursively converted into k-chordal graphs, which possess a perfect elimination scheme (PEO). Yannakakis et al. [14] demonstrated the process of converting bipartite graphs into corresponding filled chain graphs by first converting them into minimal chordal supergraphs. Furthermore, it was shown that these chain graphs have a block-contiguous elimination scheme. The idea of reduction is based on the correspondence between the perfect elimination scheme and the block-contiguous elimination scheme, which directly relates to the MCLA problem. Figure 2 illustrates the overarching concept behind the proof.

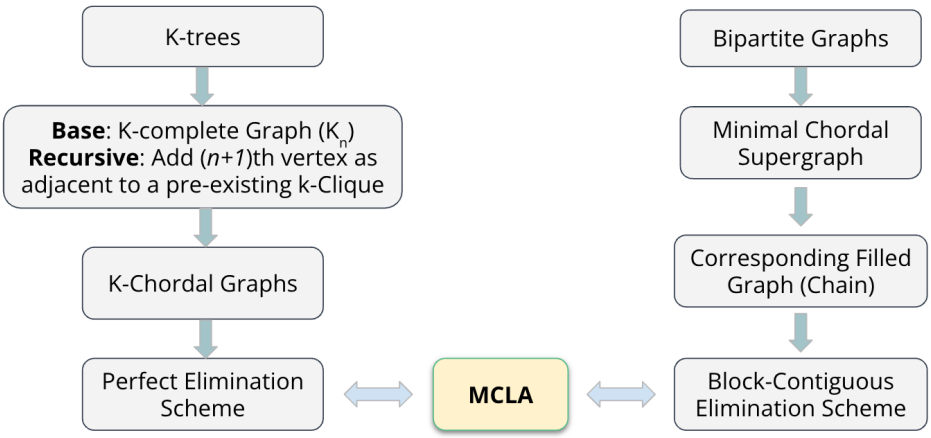


Figure 2: The overall idea behind the proof

3.3 Construction of Bipartite Graph Instance

We formalize the construction of a bipartite graph instance from a given graph $G(V, E)$ as follows:

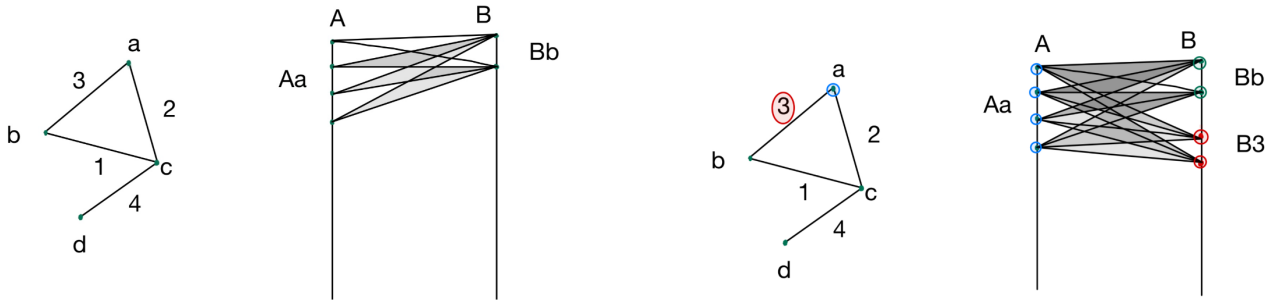
Algorithm 1 Construction of Bipartite Instance for Reduction G'

Require: A graph $G(V, E)$
Ensure: A bipartite graph $G'(A \cup B, E')$

Initialize $A, B, E' = \emptyset$
Define Nodes:
for each vertex $x \in V$ **do**
    Add $\Delta(G) + 1$ vertices to A denoted as A_x
    Add $\Delta(G) + 1 - \deg(x)$ vertices to B denoted as B_x
end for
for each edge $e \in E$ **do**
    Add 2 vertices to B denoted by B_e
end for
Define Edges:
for each vertex $x \in V$ **do**
    for each vertex in A_x **do**
        Add edges from this vertex to all vertices in B_x
        for each edge e incident to x **do**
            Add edges from this vertex to all vertices in B_e
        end for
    end for
end for

3.4 Example Construction

The construction process is illustrated with the following examples:



(a) (b)

Figure 3: First two steps of construction from given $G(V, E)$.

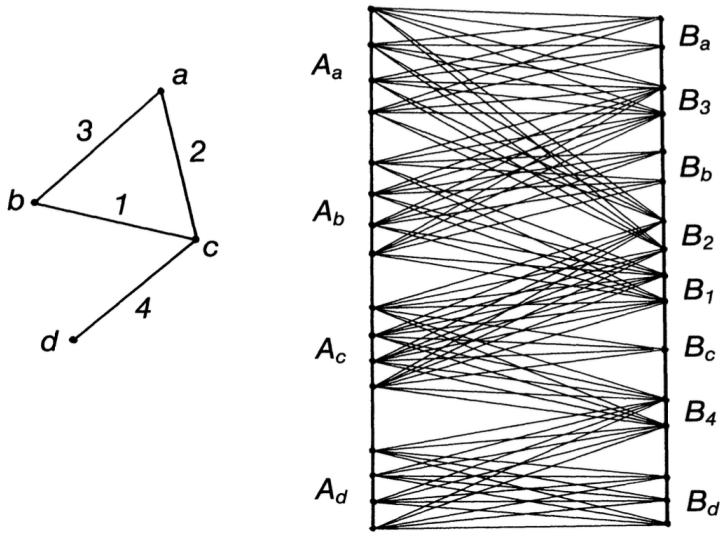


Figure 4: Final construction of the bipartite graph G'.

3.5 Formal Proof

Lemma 1. *Given a graph G and a positive integer k, G has a minimum linear cut value k with respect to some permutation π if and only if the corresponding graph $C(G')$ is a partial K'-tree for $K' = (\Delta(G) + 1)(|V| + 1) + k - 1$.*

Proof. Given that a k'-tree is a k'-chordal graph, if $C(G')$ is a partial k'-tree, then there exists a block-contiguous elimination scheme r' ensuring no vertex has a degree greater than k' upon its elimination. Let F denote the filled graph corresponding to $C(G')$ with respect to permutation r'.

- F supports a Perfect Elimination Order (PEO).
- The initiation of the PEO begins with vertices in A, following a reverse chain order.
- This ordering also adheres to a block-contiguous sequence within the blocks A_v.
- Without loss of generality, it is presumed that r' aligns with such an order. The ordering of blocks in A, induced by r', is denoted by r.

Assuming the vertices of G are sequentially numbered as per order r and identified by these numbers, we examine the graph resulting from the elimination of vertices in the initial blocks of $C(G')$. In this scenario, each vertex in A_i is adjacent to multiple vertex groups:

- The rest of $\Delta(G)$ vertices in the same block A_i.
- $\Delta(G) + 1$ vertices for un-eliminated A_j for each $j = \{i + 1, \ldots, |V|\}$.

- $\Delta(G) + 1 - \deg_G(j)$ vertices in B_j for $j = \{1, \ldots, i\}$.
- Two vertices in B_e for each edge e incident to at least one vertex in $\{1, \ldots, i\}$.

Finally, summing up all adjacencies,

$$deg(A_i) = \Delta(G) + (\Delta(G) + 1) \cdot (|V| - i) + (\Delta(G) + 1) \cdot i - \sum_{j=1}^{i} \deg_G(j) + 2|E_1^i| + 2|E_2^i|$$

Define E_1 as the set of edges with one endpoint in $\{1, \ldots, i\}$ and E_2 as the set of edges with both endpoints in $\{1, \ldots, i\}$. The simplification yields:

$$\sum \deg(j) = 2|E_2| + |E_1|$$

Simplifying further, we get the degree of A_i:

$$\deg(A_i) = (\Delta(G) + 1)(|V| + 1) - 1 + |E_1^i|$$

where $|E_1^i|$ denotes the set of edges between vertices $\{1, \ldots, i\}$ and $\{i + 1, \ldots, |V|\}$. The maximum size of $|E_1|$ corresponds to the linear cut value of G with respect to the elimination order π. Given that the instance of G has a minimum cut value k, we have $|E_1| = k$. Finally, this leads to:

$$\deg(A_i) = (\Delta(G) + 1)(|V| + 1) - 1 + k$$

Which is the maximum size of a clique in $C(G')$. Therefore, k'-chordality implies the existence of a linear arrangement with cut value k. Conversely, the existence of an ordering r with respect to which G has a linear cut value k suggests that the largest clique in F, the filled graph of $C(G')$ with respect to r', has size $k' + 1$. This completes the proof.

□

C(G') can be constructed from G in polynomial time, indicating its membership in NP. For a suitable (nondeterministic) choice of vertex order, the elimination process can be easily turned into a polynomial time verification process that confirms whether a graph is a partial k-tree.

Theorem 1. *The PARTIAL K-TREE problem (treewidth of size k), is NP-complete.*

4 Exact Algorithm

We have explored the Positive Instance Driven (PID) - Bouchitte Todinca (BT) Algorithm. This was the runner up at the Exact track of PACE 2017. This algorithm functions on the basis of minimal Seperators and finding feasible potential maximal cliques at its core. Some basic concepts are discussed below.

4.1 Historical Approach

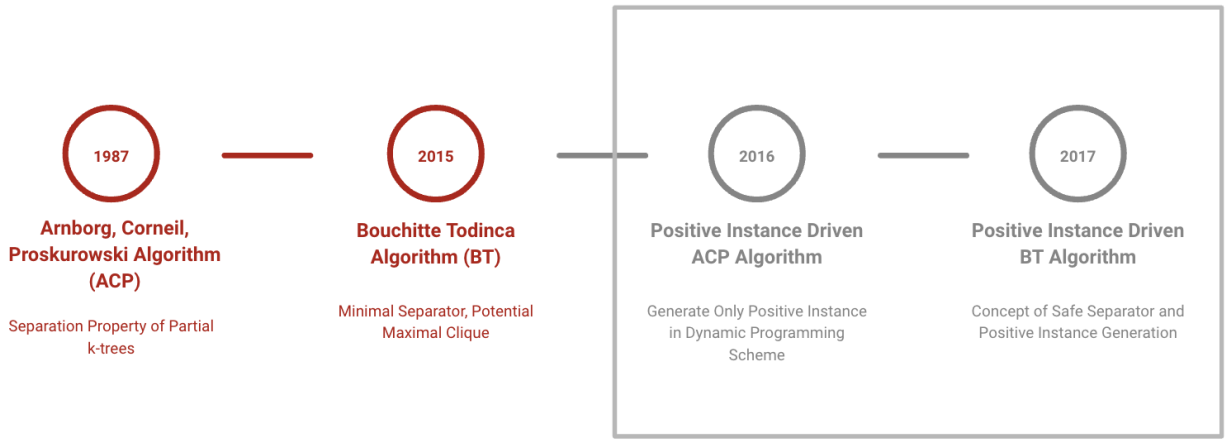
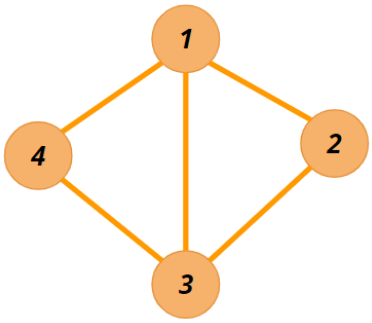


Figure 5: Timeline of key developments in the exact algorithm for treewidth computation. The illustration also includes the key ideas based on which the approach was taken.

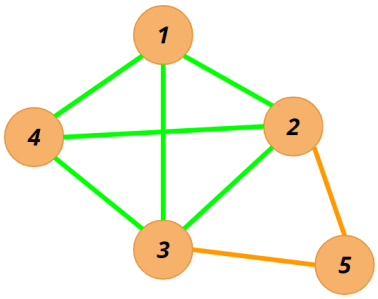
4.2 Theoretical Formulations

4.2.1 Potential Maximal Cliques:

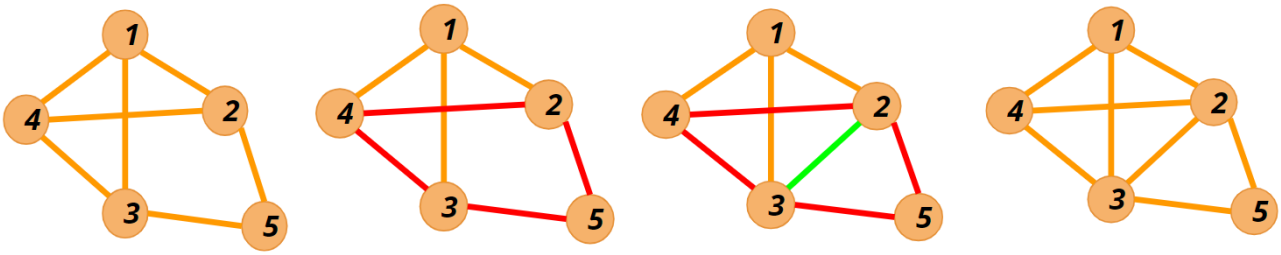
- ***Chordal Graph:*** A graph where every induced cycle has length 3 (fig: 6a)
- ***Minimal Chordal Completion:*** A chordal graph produced by adding a minimal number of edges to the original graph G. This is not unique for a graph. (fig: 6b)
- ***Potential Maximal Cliques:*** The maximal clique in any minimal chordal completion of a graph G (fig: 6c)



(a) Chordal Graph



(b) Potential Maximal Clique

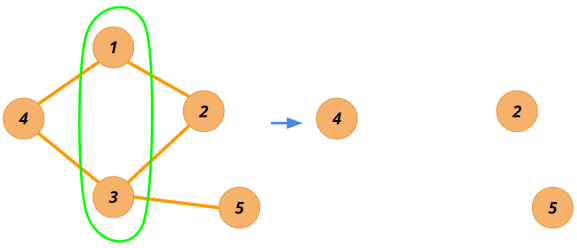


(c) Minimal Chordal Completion of a Non-chordal graph

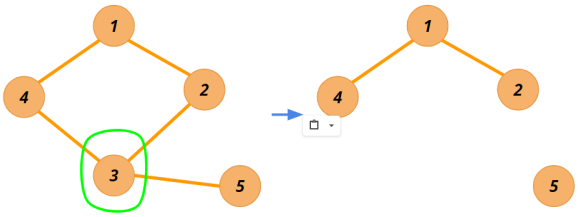
Figure 6: Definitions

4.2.2 Minimal Separators

- ***Separator:*** A set of vertices which, if removed, increases the number of cconnected components in the new graph. (fig: 7a)
- ***Minimal Separator*** A separator none of whose proper subsets is another separator. (fig: 7b)



(a) A separator



(b) A Minimal Separator

4.2.3 I-Blocks & O-Blocks

- **Observations:**
  - For each separator, total ordering of vertex separates the associated partition of $V(G)$.

  - **Inbound Partition:**
    * If some connected set C has a neighbor set $N(C)$ preceding C.
  - Otherwise, **outbound partition**.

- **Facts:**
  - For any inbound vertex set C, neighbor of C, $N(C)$ is a *minimal separator*.

- **Formally,**
  - A pair of $(N(C), C)$ defined as a full-block is an **I-block** if C is inbound and $|N(C)| \leq k$.
  - A pair of $(N(C), C)$ defined as a full-block is an **O-block** if C is outbound and $|N(C)| \leq k$.

4.2.4 Feasible Potential Maximal Clique

- Define $G[C]$, $C \subseteq V(G)$, as a graph from taking the closed neighbourhood of C and completing $N(C)$, neighbours of C a clique.

- **Feasible:**
  - A vertex set C is feasible if treewidth$(G[C]) \leq k$.

- **Support**(C):
  - The set of partitions arising from maintaining C as a separator, that has partial coverage.
  - That is, some subset of the separated vertex set is connected to only $T \subset C$ (proper subset of C).

- A Potential Maximal Clique Ω is feasible if,
  - $|\Omega| \leq k + 1$
  - For every $C \in$ support(Ω) is feasible.

4.3 Bouchitté–Todinca Algorithm

The BT algorithm focuses on computing the treewidth of a given graph $G(V, E)$ alongside a specified parameter K. It systematically identifies the minimal separator set, derives potential maximum clique sets based on these separators, and utilizes them to infer the tree decomposition, culminating in the determination of the treewidth.

Algorithm 2 BT Algorithm for Treewidth Verification

Require: A graph $G(V, E)$ and an integer k
Ensure: Returns TRUE if $tw(G(V, E)) \leq k$, otherwise FALSE
1: Find the Minimal Separator Set
2: Derive the Potential Maximum Clique set based on the Associated Minimal Separator set.
3: Infer the tree decomposition using Cuts using the Maximum Clique set.
4: **if** $tw(G(V, E)) \leq k$ **then**
5: **return** TRUE
6: **else**
7: **return** FALSE
8: **end if**

4.3.1 Weakness of Bouchitté–Todinca Algorithm

Following lists the improving scopes of Bouchitté–Todinca algorithm:

1. Exhaustive checking of each vertex neighbor while searching for a Minimal Separator.
2. Weak Lower-Upper Bound gap for unsolved DIMACs graph-coloring instances.
3. Negligence of Safe Separators.

Safe separators is a set of vertices $S \subseteq V$ that is a separator of G, and the treewidth of G equals the maximum treewidth over all connected components W of $G - S$, where the graph obtained by making S a clique in the subgraph of G, induced by $W \cup S$, preserves the treewidth. A critical study on safe separators for treewidth is provided by Hans L. Bodlaender and Arie M.C.A. Koster [15].

4.4 Positive Instance Driven Bouchitté–Todinca Algorithm (PID-BT)

In the dynamic programming approach, the search space can be reduced by beaming the search towards positive instances only. This idea was first incorporated in Arnborg, Corneil, and Proskurowsky (ACP) algorithm. Naturally, the next approach was to reduce the search space of BT-algorithm incorporation PID. The following figure accumulates the idea:

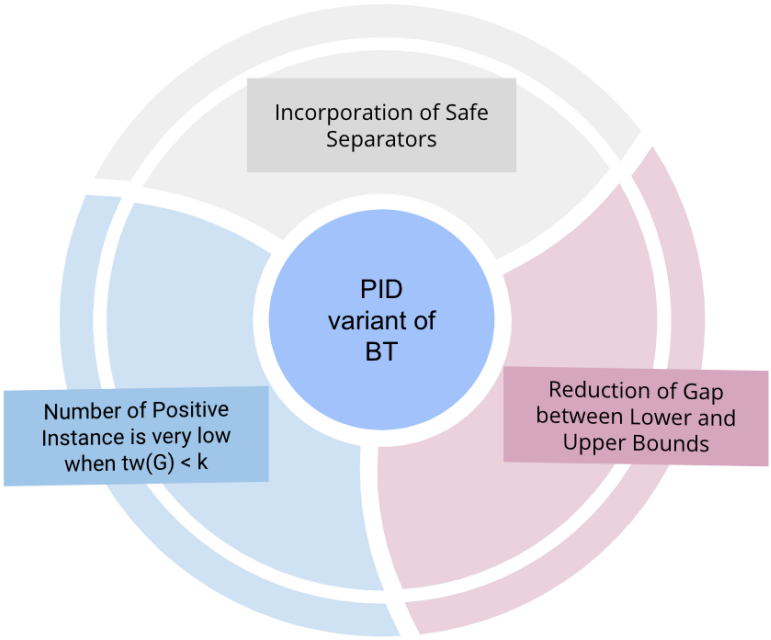


Figure 8: Improvements in PID-BT algorithm

4.4.1 Modification of Recurrence for PID Variant

One of the challenging aspects was to modify the recurrence relation used in the dynamic programming for finding the set of minimal separators. Firstly, the connected subset of vertices under consideration for minimal separator is exponential in size. Thus, this may involve indefinite number of vertex subset search space. The challenges were faced followingly:

- Incorporate a total ordering of the vertex set.
- Consider only the minimal separators that are inbound and outbound, which are stored in I-blocks and O-blocks data structures.
- This approach narrows down to identifying the feasible potential maximum clique.
- The identified clique is then recursively fed into the next step of the dynamic programming process.

4.4.2 PID-BT Algorithm

Algorithm 3 PID-BT Algorithm

Require: A graph $G(V, E)$ and a positive integer k
Ensure: "YES" if $tw(G) \leq k$; "NO" otherwise

Let $I_0 = \emptyset$ and $O_0 = \emptyset$.
Initialize P_0 and S_0 to $\emptyset$.
Set $j = 0$.
for each $v \in V(G)$ **do**
  if $N[v]$ is a potential maximal clique with $|N[v]| \leq k + 1$ **then**
    Add $N[v]$ to P_0 and if $support(N[v]) = \emptyset$ then
    Add $N[v]$ to S_0.
    if $outlet(N[v]) \neq \emptyset$ **then**
      Let $C = crib(outlet(N[v]), N[v])$ and if $C \neq C_h$ for $1 \leq h \leq j$ then
      Increment j and let $C_j = C$.
    end if
  end if
end for
Set $i = 0$.
repeat
  While $i < j$, do the following:
  Increment i and let $I_i = I_{i-1} \cup \{C_i\}$.
  Initialize O_i to O_{i-1}, P_i to P_{i-1}, and S_i to S_{i-1}.
  for each $B \in O_{i-1}$ such that $C_i \subseteq B$ and $|N(C_i) \cup N(B)| \leq k + 1$ **do**
    Let $K = N(C_i) \cup N(B)$
    if K is a potential maximal clique **then**
      Add K to P_i.
      if $|K| \leq k$ and there is a full component A associated with K **then**
        Add A to O_i.
      end if
    end if
  end for
  For each $A \in O_i \setminus O_{i-1}$ and $v \in N(A)$, let $K = N(A) \cup (N(v) \cap A)$ and if $|K| \leq k + 1$ and K is a potential maximal clique then add K to P_i.
  for each $K \in P_i \setminus S_{i-1}$ **do**
    if $support(K) \subseteq I_i$ **then**
      Add K to S_i and if $outlet(K) \neq \emptyset$ then
      Let $C = crib(outlet(K), K)$ and if $C \neq C_h$ for $1 \leq h \leq j$ then
      Increment j and let $C_j = C$.
    end if
  end for
until j is not incremented during the iteration step
if there is some $K \in S_j$ such that $outlet(K) = \emptyset$ **then**
  return "YES"
else
  return "NO"
end if

4.4.3 Breaking Down the Algorithm

The algorithm can be decomposed into four distinct steps:

1. Subproblem Generation and I-block Initialization: Here, the sets of inbound minimal separators are produced which are the candidates for potential maximal clique. The candidates are the associated connected subsets of the separators.

2. Deriving Maximal Potential Cliques: From the I-blocks and O-blocks, the maximal potential cliques are aggregated. Here, sets with cardinality no more than k+1 is listed.

3. Checking Feasibility of Cliques: To reduce the size of the search space, the sets of maximal potential cliques are filtered by feasibility criteria. This is the PID-incorporation i.e., only feasible maximal clique which are most likely to produce the bags of optimal treewidth, are only kept to further climb up in the bottom-up dynamic programming approach.

4. Making Decision: The treewidth of a given graph $G(V, E), tw(G) \leq k$ if and only if all components associated are feasible.

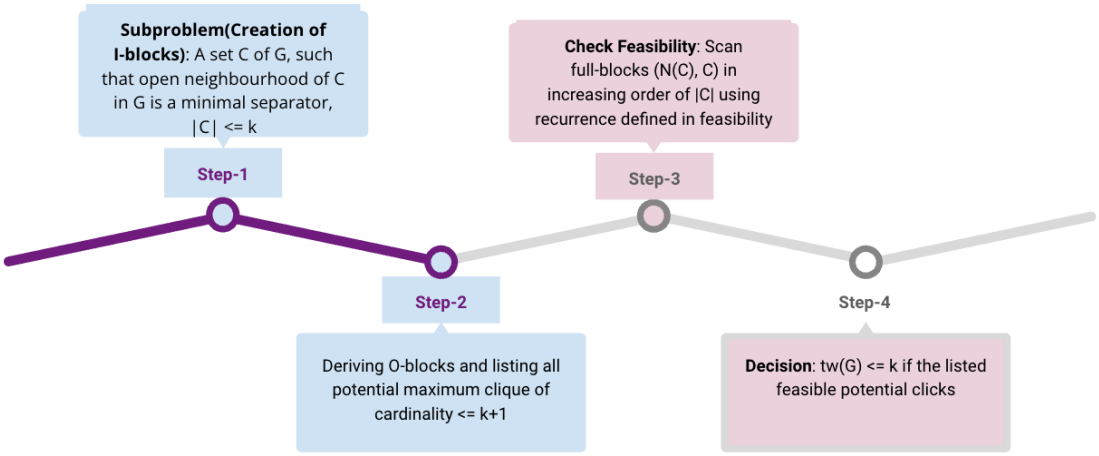


Figure 9: First part of the broader timeline of the PID-BT algorithm.

4.4.4 Correctness of PID-BT Algorithm

Theorem 2. *The treewidth of a given graph $G(V, E)$, $tw(G) \leq k$ if and only if G has a feasible potential maximal clique Ω with a non-full associated outbound component (all components associated are feasible).*

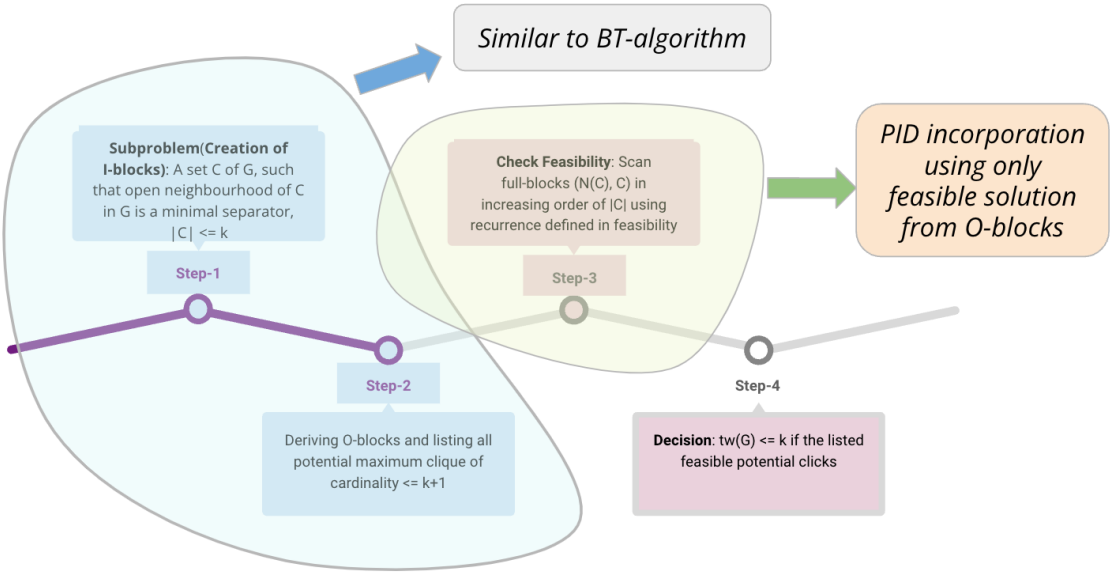


Figure 10: Annotation of the steps of the algorithm. The colored section shows step-3 incorporates the PID in the search space.

The theorem is inferred from [4]. The proof of this theorem relies on induction on the recurrence relation presented within the same paper. The induction steps shows that, on each step, following conditions are always satisfied:

1. For every $1 \leq h \leq i$, $(N(C_h), C_h)$ is a feasible I-block.
2. $I_i = \{C_h \mid 1 \leq h \leq i\}$.
3. For every $A \in O_i$, $(N(A), A)$ is a feasible O-block.
4. Every $K \in P_i$ is a buildable potential maximal clique.
5. Every $K \in S_i$ is a feasible potential maximal clique.

Thereby establishing that the PID-BT algorithm correctly computes the treewidth of a given graph.

4.4.5 Runtime Analysis

The runtime of this algorithm has the following closed form: $O(I\text{-block}^k \cdot O\text{-block}^k)$

5 Implementation of PID-BT Algorithm

5.1 Dataset Used

We use the graph instances that were used in The Second Parameterized Algorithms and Computational Experiments Challenge (PACE 2017). Half of the instances were made public before the challenge and the other half remained hidden until the conclusion of the challenge. More details on the datasets can be found in the **Official Report**.

There are 200 benchmark instances, labeled ex001.gr to ex200.gr. Larger numbers in the filename should (as a rule of thumb) correspond to harder instances. The odd instances are public and the even instances are secret. All instances are based on real-world data, but about 50 were boosted in hardness using a natural random process.

5.2 Results for the PID-BT algorithm

Here we present the execution times, number of bags generated, and the treewidth with respect to the number of nodes and number of edges for the PID-BT algorithm.

5.2.1 Execution Time

The execution time graph is provided in figure 11 and figure 12. We see that when the number of nodes and the number of edges are less, the execution time is more. There is a logical reason behind it. For that we need to understand the relation of the treewidth with number of nodes and edges. We will discuss it in the next part.

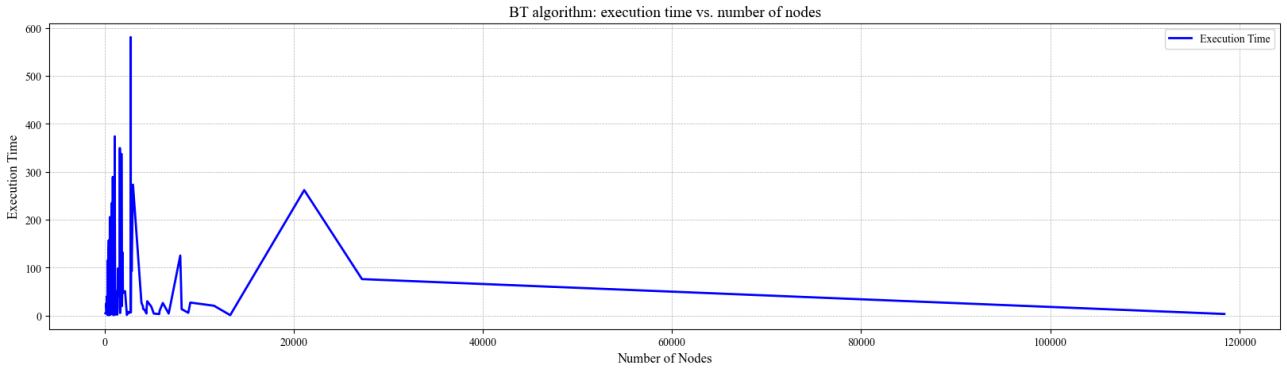


Figure 11: Execution Time vs. Number of Nodes (Bouchitte-Todinca Algorithm)

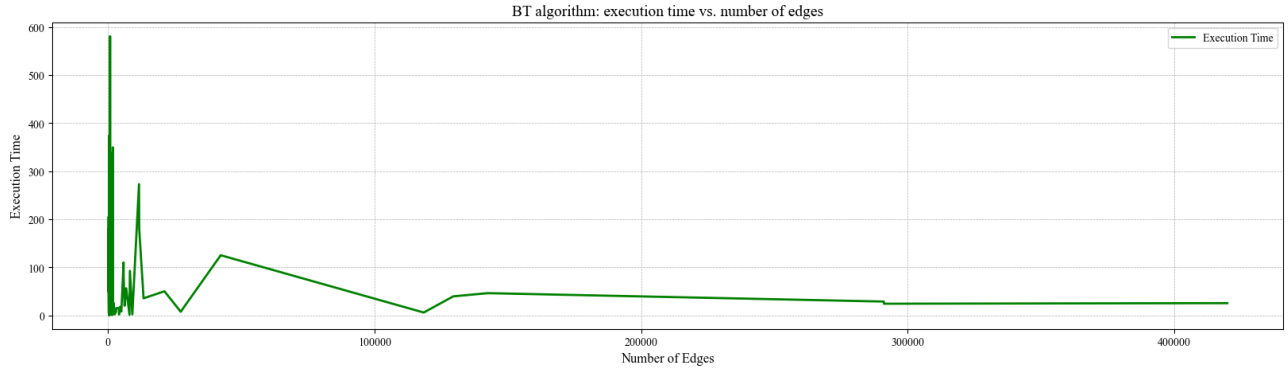


Figure 12: Execution Time vs. Number of Edges (Bouchitte-Todinca Algorithm)

5.2.2 Number of Bags

Total number of bags is larger when there are less nodes and less number of edges.

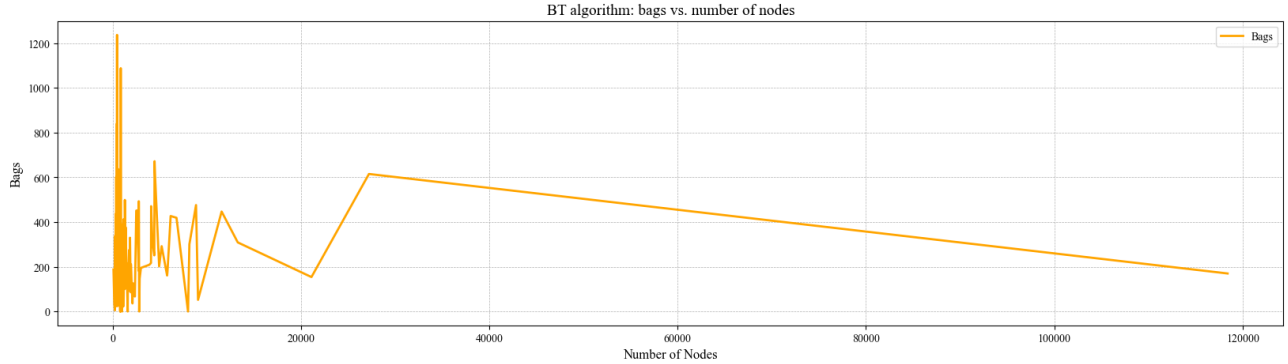


Figure 13: Number of Bags vs. Number of Nodes (Bouchitte-Todinca Algorithm)

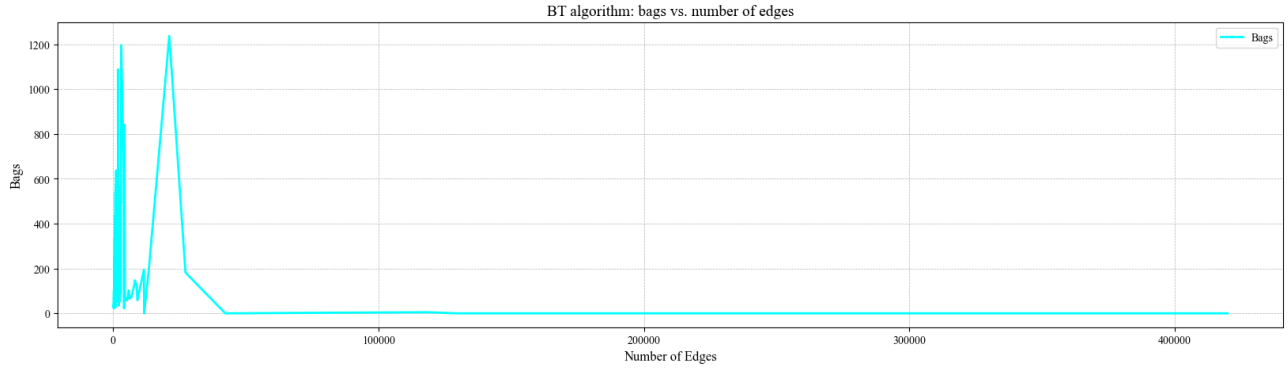


Figure 14: Number of Bags vs. Number of Edges (Bouchitte-Todinca Algorithm)

5.2.3 Treewidth

When the number of nodes are small, there is a chance that the graph is highly connected. Which means that in each bag the number of nodes will be higher. As a result the treewidth will be larger, as we can see in figure 15. Moreover, when the number of edges increase, it means that the connectivity of the graph is increasing which in turn increase the treewidth which we can see in figure 16.

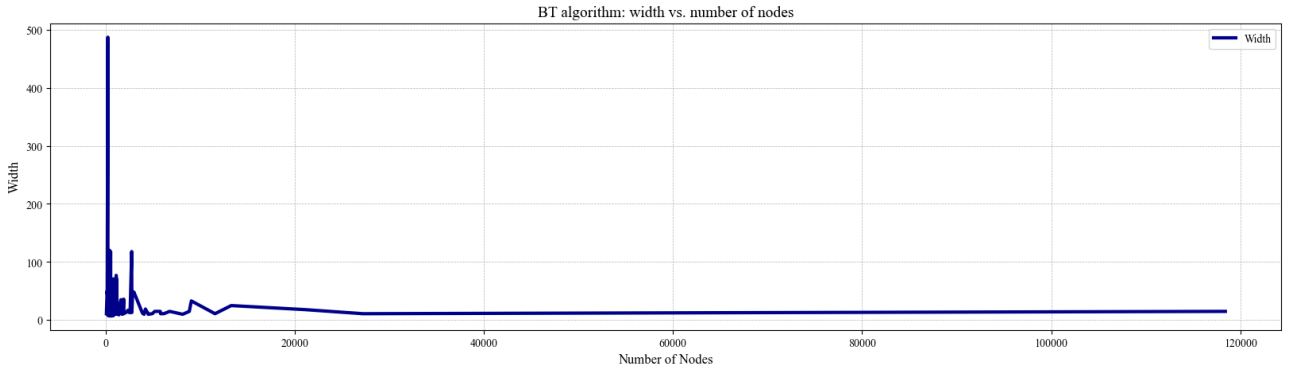


Figure 15: Width vs. Number of Nodes (Bouchitte-Todinca Algorithm)

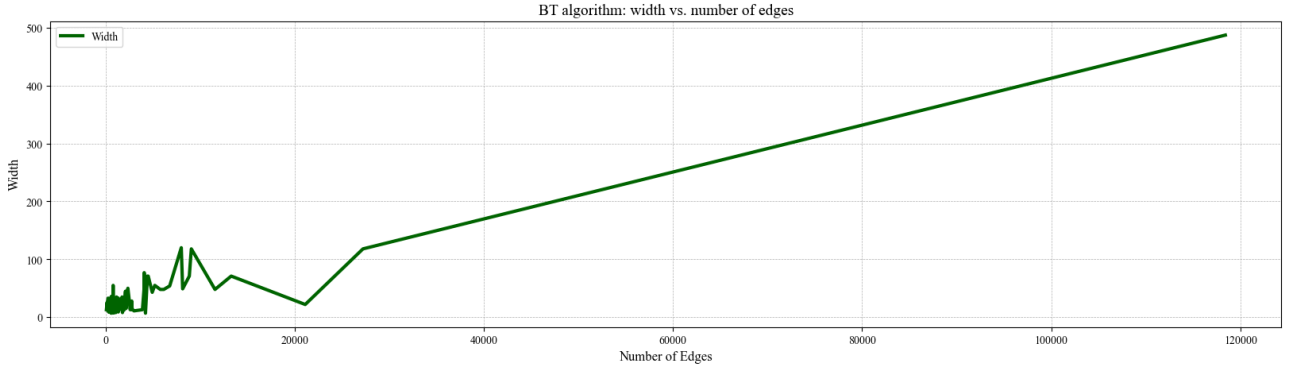


Figure 16: Width vs. Number of Edges (Bouchitte-Todinca Algorithm)

6 Min Degree Heuristic

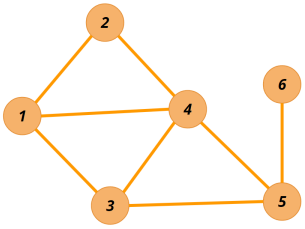
In this section, we will move forward to a simple heuristic algorithm which is very less complex than the exact algorithm in terms of approach and running time. We will also compare its performance with the exact algorithm.

The basic idea is simple here. We will pick the node with the lowest degree and, along with that node, put that node's neighbors into a bag. Then we remove that node. This goes on until we find a clique or complete graph. The algorithm is discussed in the following.

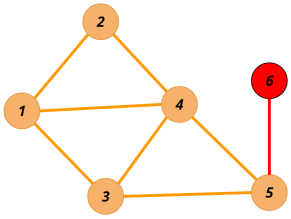
Algorithm 4 Tree Decomposition Algorithm

Require: A graph $G = (V, E)$
Ensure: Tree Decomposition of G
$T \leftarrow$ an empty tree
while $|V| > 1$ **do**
  Find $u \in V$ with the minimum degree
  Create a bag B with u and its neighbors $N(u)$
  for each pair of non-adjacent neighbors $(v, w) \in N(u) \times N(u)$ **do**
    Add edge (v, w) to E
  end for
  Remove u and its incident edges from G
  Update the degrees of nodes in V
  Add bag B as a node in T
  if all remaining nodes in V form a clique **then**
    break
  end if
end while
Add remaining nodes as a bag in T
return T

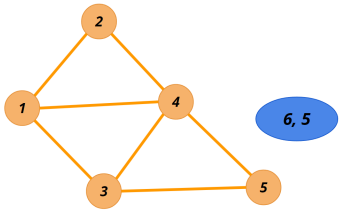
In the figure **??**, we provide a demonstration of a simple simulation to find a tree composition using the min-degree heuristic algorithm.



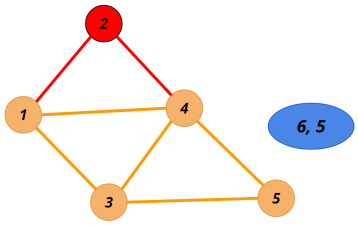
(a) Given Graph



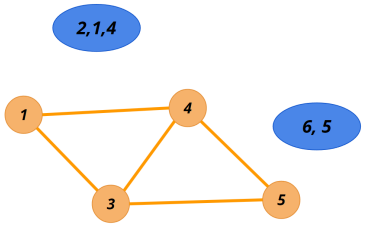
(b) Min Degree Node: 6



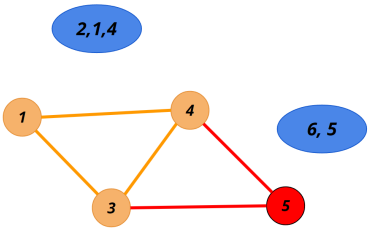
(c) Include 6 and 3 into the bag



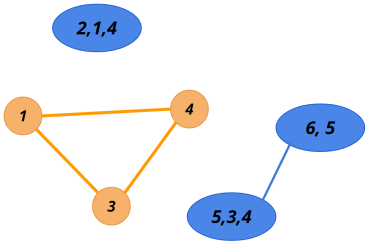
(d) Min Degree Node: 2



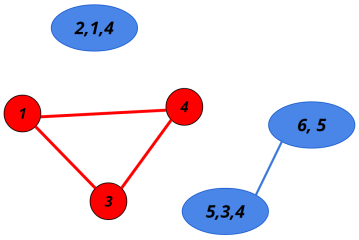
(e) Include 1,2,4 into the Bag



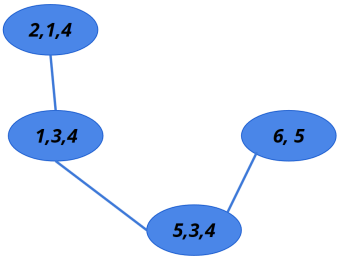
(f) Min Degree Node: 5



(g) Include 3,4,5 into the Bag



(h) We find a complete graph here



(i) Include 1,3,4 in a Bag and join the nodes

Figure 17: Simulation Results for the Min Degree Heuristic

6.1 Results for the Heuristics algorithm

Here we present the execution times, number of bags generated, and the treewidth with respect to the number of nodes and number of edges for the Min degree heuristic algorithm.

6.1.1 Execution Time

Similar to as the PID-BT algorithm, the min degree heuristic based algorithm shows a similar plot for the execution time with respect to the number of nodes. As the number of nodes are less, the graph tends to be more connected hence the treewidth is large. Therefore calculation of the treewidth takes a larger time. However, the execution time does not show any regular pattern with respcect to the number of edges.

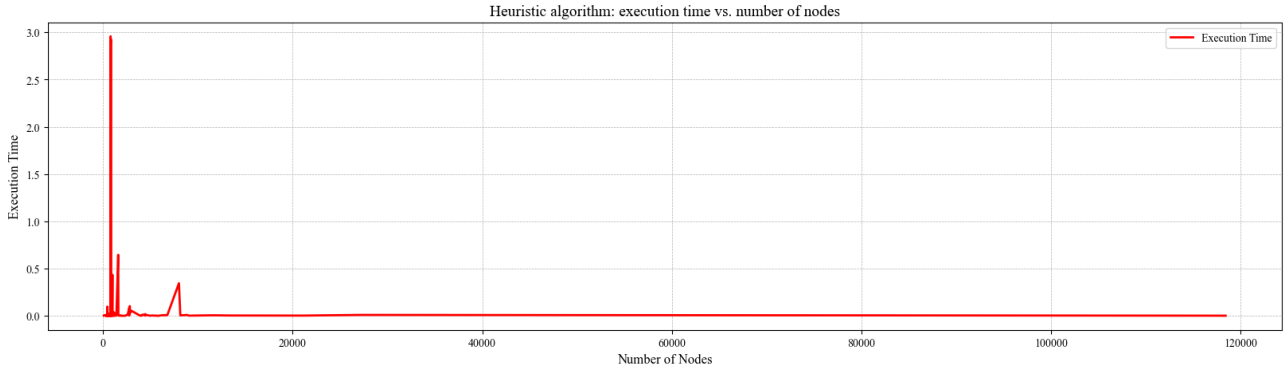


Figure 18: Execution Time vs. Number of Nodes (Minimum Degree Heuristic)

Figure 19: Execution Time vs. Number of Edges (Minimum Degree Heuristic)

6.1.2 Number of Bags

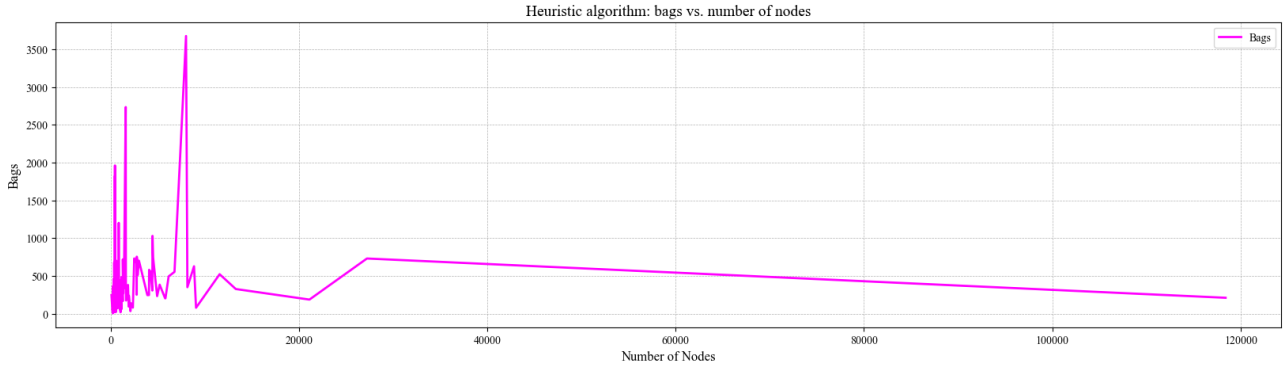


Figure 20: Number of Bags vs. Number of Nodes (Minimum Degree Heuristic)

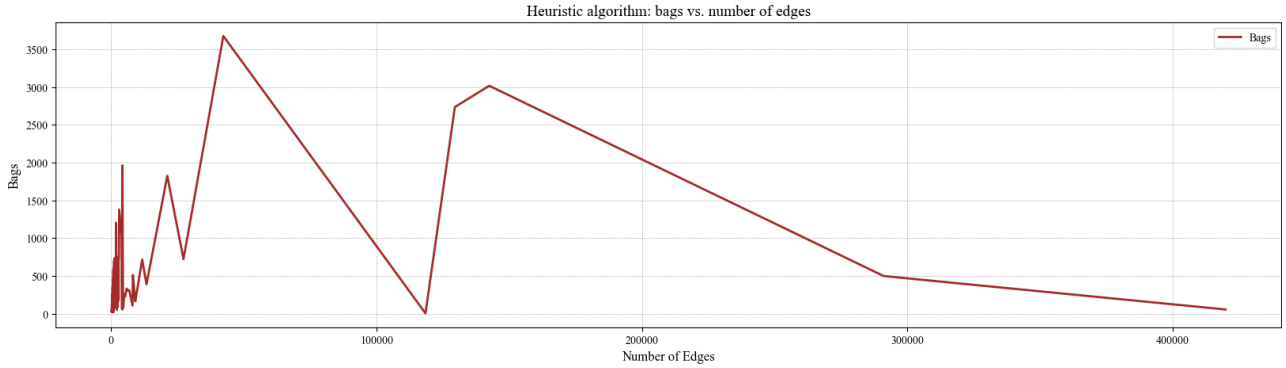


Figure 21: Number of Bags vs. Number of Edges (Minimum Degree Heuristic)

6.1.3 Treewidth

Similar to as PID-BT algorithm, the number treewidth is directly dependent on the number of nodes and number of edges. As the number of nodes increase, the graph become more sparse, hence reducing the number of nodes per bag. Moreover, increasing the number of edges increase the graph's connectivity which increases the treewidth of the graph.

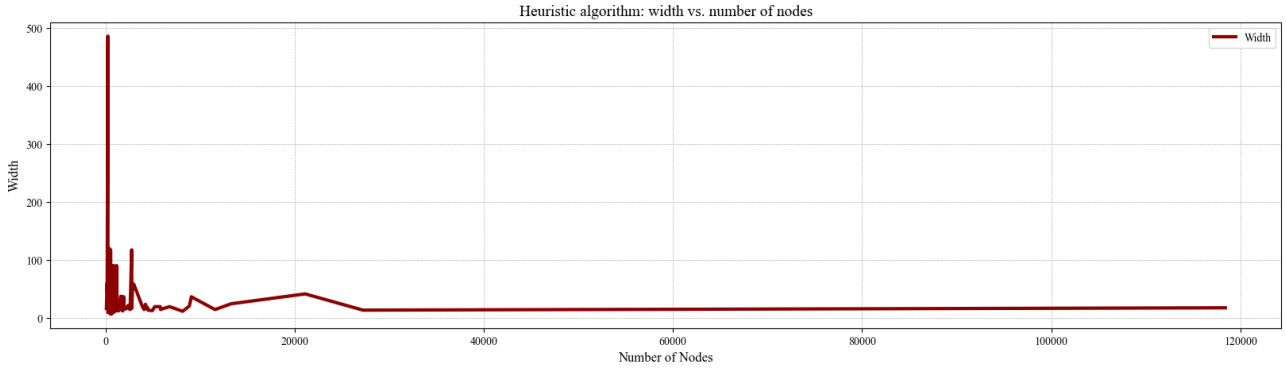


Figure 22: Width vs. Number of Nodes (Minimum Degree Heuristic)

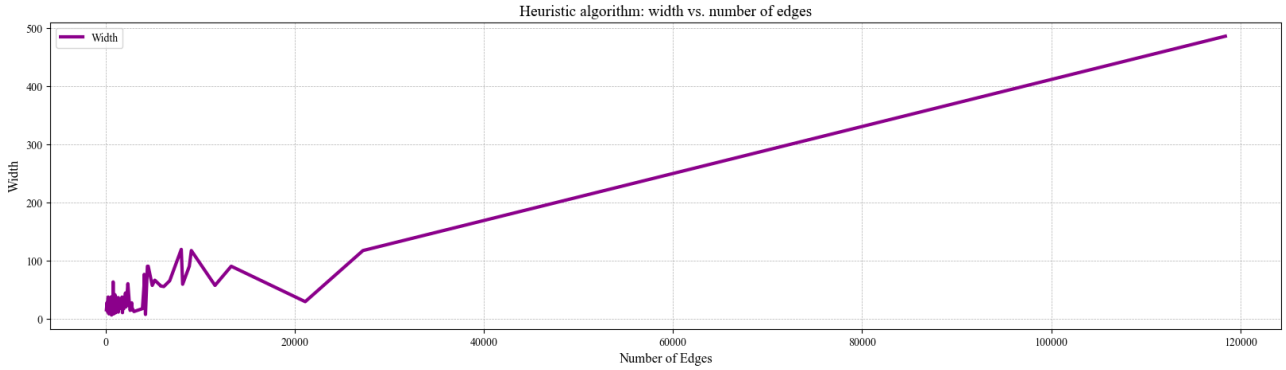


Figure 23: Width vs. Number of Edges (Minimum Degree Heuristic)

6.2 How Good the Heuristic Performs?

We compare the performance difference between the PID-BT algorithm and the heuristic algorithm by taking the ratio between them for difference instances. In figure 24, we compare the performances of the exact and the heuristic algorithm with varied number of nodes. We see that, the approximation ratio is between 1.10 to 1.74. With increasing number of nodes, the performance curve aligns perfectly.

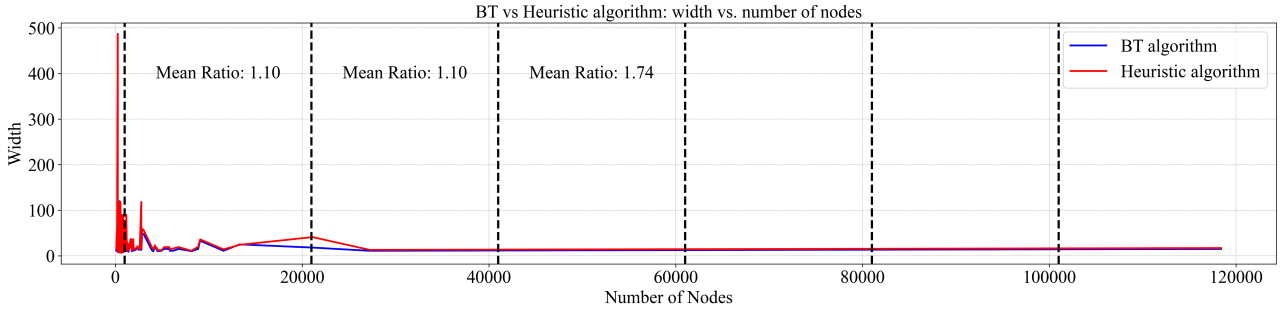


Figure 24: Comparison of the Performance Between BT and the Min Degree Heuristic with respect to Nodes

In figure 25, we compare the performances with respect to the number of edges. We see that, initially, the approximation ratio is around 1.13. However, with increasing number of edges the approximation ratio gets even better down to 1.11. Its notable that, the execution time is very less for the approximation algorithm compared to the PID-BT algorithm. In the PID-BT algorithm, the maximum execution time is around 600s,whereas in the heuristic algorithm, the maximum execution time is around 3s. This is the reason of the widespread use of the min degree approximation algorithm in real life, while dealing with problems related to treewidth.

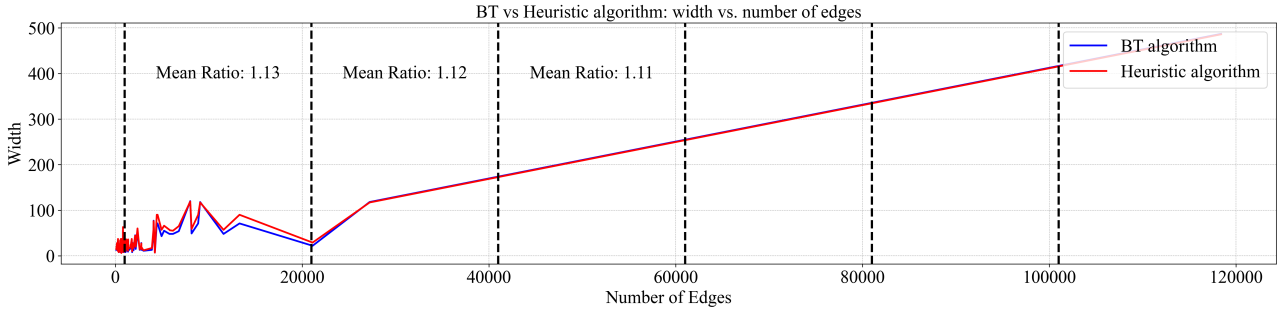


Figure 25: Comparison of the Performance Between BT and the Min Degree Heuristic with respect to Edges

7 Heuristic Algorithm(Flow Cutter)

For heuristic algorithm, we have selected the Flow Cutter [16], which won the 2nd place in the heuristic section. We picked this algorithm due to the availability of a paper associated with it [9]. Also the core algorithm uses a familiar concept, the Max Flow Min Cut theorem.

The Flow Cutter algorithm performs tree decomposition with the *nested dissection* method. The novelty of Flow Cutter is an algorithm, called "Core Flow Cutter" that computes small balanced graph cuts. We first describe the "Core Flow Cutter" algorithm. Then we describe the *nested dissection* method, which utilizes the cut algorithm.

7.1 Definitions and Terminologies

s-t cut In graph theory, an s-t cut (or source-sink cut) of a graph $G = (V, E)$ is defined as a partition (S, T) of the vertex set V into two disjoint sets S and $T = V \setminus S$, such that the source vertex s is in S and the sink vertex t is in T. A cut is called **balanced** if $|S| \approx |T|$. An edge $(u, v) \in E$ is called an cut edge if $u \in S, v \in T$ or $u \in T, v \in S$. let C be the set of all such edges. The **size** of a cut is defined as the cardinality of C, that is $|C|$. A **small balanced** cut is one that is balanced and its size is small. We illustrate these in figure 26.

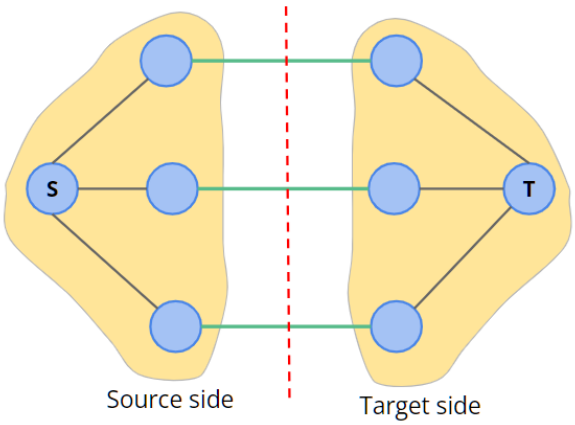


Figure 26: Example of a small balanced st-cut. The green edges are the cut edges. Both side of the cut contain equal number of vertices. The cut size is 3, which is the minimum size of a balanced cut in this graph.

Unit flow A flow network where each edge has unit capacity and amount of flow through an edge is either 0 or 1.

Saturated edge an edge with flow 1

Unsaturated edge an edge with flow 0

Perfect Elimination Ordering Consider a graph with a set of vertices, denoted as $G(V, E)$ where V is the set of vertices and E is the set of edges.

- The ordering of the vertices V is such that for each vertex v, the set of neighbors of v that appear later in the order form a clique.
- For each vertex v, we define:
  - S as the set that contains the vertex v and all its neighbors that come after it in the ordering, i.e., $S = \{v\} \cup N_j(v)$ where $N_j(v)$ represents the neighbors of v with position after v in the ordering.
  - The set S should form a clique.

An example of such an ordering for the graph in Figure 27 is: $a, b, c, d, e, f, g, h, i, j$. For instance, take the vertex f. The neighbors of f that appear to the right are: i and j. Therefore, f, i, and j forms a clique.

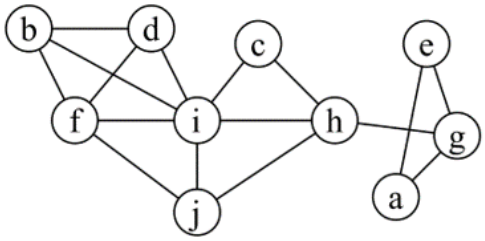


Figure 27: An example graph with vertices labeled to illustrate Perfect Elimination Ordering.

Small Balanced Node Separator A Small Balanced Node Separator is a subset of vertices in a graph whose removal results in the division of the graph into two or more disconnected subgraphs. The goal is to have these subgraphs be approximately equal in size, and the separator should be as small as possible. This problem is NP-hard for general graphs. Notably, a graph of treewidth k possesses a balanced separator with no more than $k + 1$ vertices.

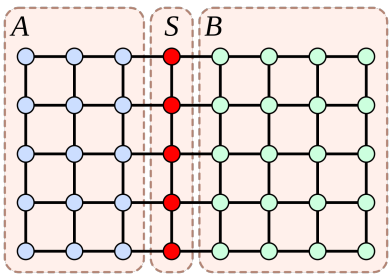


Figure 28: The separator nodes S divide the graph into subgraphs A and B.

Chordal Supergraph A chordal completion or chordal supergraph of a given undirected graph G is a chordal graph, on the same vertex set, that has G as a subgraph. Transforming a non-chordal graph into a chordal graph involves adding the minimum number of edges necessary to ensure that all cycles of four or more vertices have a chord (triangulated).

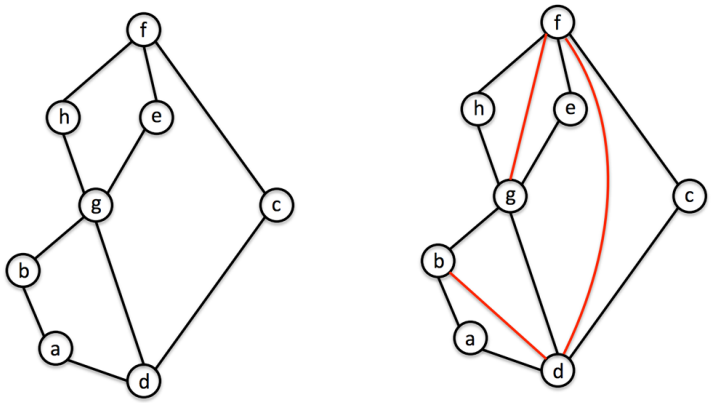


Figure 29: Chordal completion of a graph shown on the right

Maximum Spanning Tree A Maximum Spanning Tree (MST) of a weighted graph is a spanning tree in which the sum of the weights of the edges in the tree is maximized. Let $G = (V, E)$ be a connected, undirected graph with a weight function $w : E \to \mathbb{R}$ associated with its edges. A maximum spanning tree is a spanning tree $T = (V, E_T)$ for which the sum of the weights $w(e)$ of all the edges $e \in E_T$ is maximized, formally:

$$\max_{T \subseteq E} \sum_{e \in T} w(e)$$

where T ranges over all the spanning trees of G.

7.2 Core Flow Cutter algorithm

The Core Flow Cutter algorithm computes small balanced st-cut (defined in 7.1).It considers unit flow in the graph and utilizes the max flow min cut theorem.

7.2.1 High level overview

The algorithm starts with a given source and target. It then computes a cut using the max flow min cut theorem. It then manipulates the cut by increasing the number of sources or targets. While doing so, it tries to keep the net flow minimum so that cut size is as small as possible.

7.2.2 Algorthm in details

We have presented the pseudocode of the algorithm in 5. The algorithm takes an undirected graph G, a source vertex s and a target vertex t as input. It returns a balanced cut that is expected to be small. We now explain the algorithm line by line.

Algorithm 5 Core Flow Cutter

Require: An undirected graph $G(V, E)$, a source $s \in V$, a target $t \in V$
Ensure: A small balanced cut (S, T)
1: Initialize S and T as empty sets: $S \leftarrow \emptyset$, $T \leftarrow \emptyset$
2: $S \leftarrow S \cup \{s\}$, $T \leftarrow T \cup \{t\}$
3: **repeat**
4: $(S', T') \leftarrow \textsc{MaxFlowMinCut}(G, S, T)$
5: **if** $|S'| < |T'|$ **then**
6: $S \leftarrow S \cup S'$
7: $piercingNode \leftarrow \textsc{SelectPiercingNode}(G, S', T', s, t)$
8: $S \leftarrow S \cup \{piercingNode\}$
9: **else if** $|S'| > |T'|$ **then**
10: $T \leftarrow T \cup T'$
11: $piercingNode \leftarrow \textsc{SelectPiercingNode}(G, S', T', s, t)$
12: $T \leftarrow T \cup \{piercingNode\}$
13: **else**
14: $S \leftarrow S \cup S'$
15: $T \leftarrow T \cup T'$
16: **end if**
17: **until** $|S| + |T| = |V|$
18: **return** (S, T)

In line $1 - 2$, we initialize S and T as sets containing s and t respectively. These two will contain the final cut that the algorithm returns. Then we keep iterating until S and T contains a valid cut (line $3 - 17$). At the beginning of each iteration, we use the Max Flow Min Cut theorem to compute a graph cut, where sources $\in S$ and targets $\in T$ (line 4). Then, we check which side of the cut has less nodes (line $5, 9$). If the source side has less nodes, then we convert all the source side nodes to sources (line 6). We also consider an additional vertex, termed *piercing node* and convert it to a source (line 7). This node is chosen from the vertices of the opposite side that are attached to a cut edge. Such nodes are termed *candidate piercing nodes*. The piercing node is added to guarantee that we get a new cut from the Max Flow Min Cut theorem in the next iteration. There are 2 heuristics to choose a *piercing node* from the candidates.

Primary Heuristic: Choose a node from the candidates such that no path of unsaturated edges exists from the chosen node to any of the target nodes. This is because if the chosen node contains a path of saturated edges, then making it a source vertex will increase the net flow by

1. This will increase the cut size.

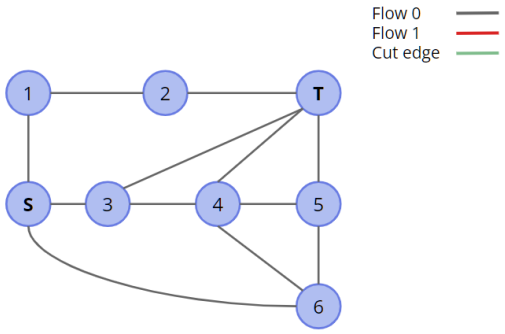
Secondary Heuristic: This is used to break ties after applying primary heuristic. let $d(u, v)$ be the shortest distance between vertex u and v. Select a node p, such that $d(p, t) - d(p, s)$ is maximized.

Algorithm 6 Select Piercing Node

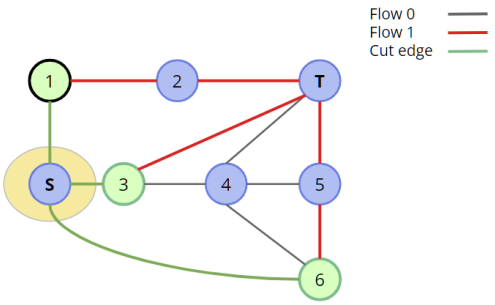
Require: An undirected graph $G(V, E)$, a cut (S, T), original source s, original target t
Ensure: A piercing node following primary and secondary heuristic

```
1: Initialize C as empty sets: C ← ∅
2: for all edges (u, v) ∈ E do
3:    if u ∈ S and v ∈ T then
4:       C ← C ∪ (u, v)
5:    end if
6: end for
7: Initialize p, q as null: p ← null, q ← null
8: Initialize pd, qd: pd ← −∞, qd ← −∞
9: for all edges (u, v) ∈ C do
10:   d ← minimum distance from v to t
11:   if there is a path consisting of saturated edges from v to any member of T then
12:      if d > pd then
13:         pd ← d
14:         p ← v
15:      end if
16:   else
17:      if d > qd then
18:         qd ← d
19:         q ← v
20:      end if
21:   end if
22: end for
23: if pd > −∞ then
24:   return p
25: end if
26: return q
```

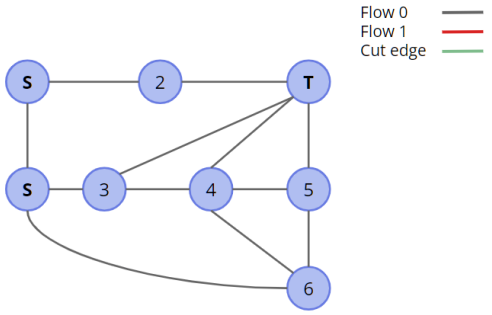
The algorithm 6 uses primary and secondary heuristics to select a piercing node from the candidate piercing nodes. We use this algorithm in line 7 of the Core Flow Cutter algorithm to select a piercing node. Then we add it to S (line 8). The same things are done in lines $10 - 12$ as in lines $6 - 8$, but for the target side rather than the source side. In each iteration, we get a new cut and the $|S|$ or $|T|$ increases. At one point, either we find a balanced cut (line $14 - 15$) or (S, T) becomes a valid cut. Then we return (S, T) as found cut.We illustrate the working of the algorithm in figure 30.



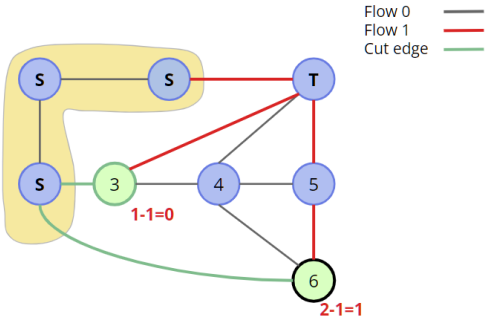
(a) Input Graph to the Core Flow Cutter algorithm. The source node is marked by **S**, and the target node is marked by **T**



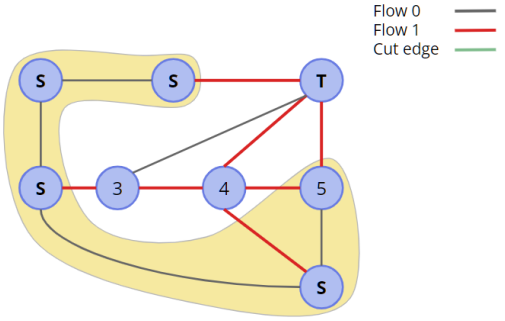
(b) Selecting piercing node in 1st iteration. The candidates nodes are marked with green fill. Among them, 1 is selected in accordance to primary heuristic.



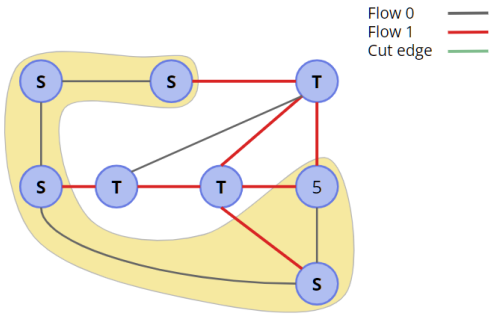
(c) S and T after 1st iteration. The selected piercing node, node 1, has been converted to a source.



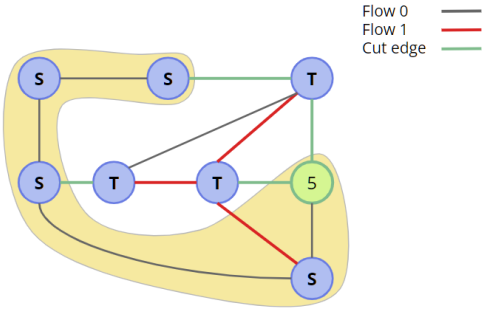
(d) Selecting piercing node in 3rd iteration.All candidates have path with non saturated edges to a target node. Secondary heuristic is used to break ties. Node 6 is selected.



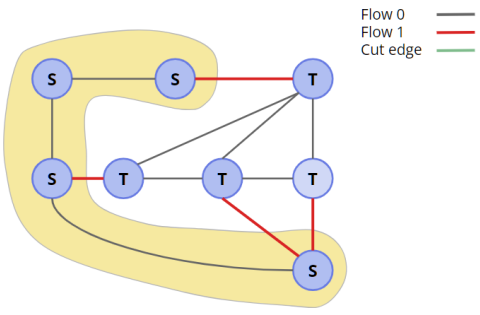
(e) Graph cut from Max Flow Min Cut in 4th iteration. Now the target side has lesser nodes.



(f) The target side nodes have been converted to targets.



(g) Selecting piercing node in 4th iteration



(h) Balanced cut found in 5th iteration

Figure 30: Example simulation of the Core Flow Cutter algorithm.

7.2.3 Finding General Cuts

To find a small balanced general cut, we run the Core Flow Cutter algorithm q times, while taking st pairs randomly from uniform distribution. With sufficient large q, we have a very high probability of finding a good cut. The Flow Cutter paper recommends $q = 20$.

7.2.4 Finding node separators

To find a node separator from the cut algorithm, we transform a given undirected graph $G = (V, E)$ to a directed graph $G' = (V', E')$. The process of transforming G to G' is as follows

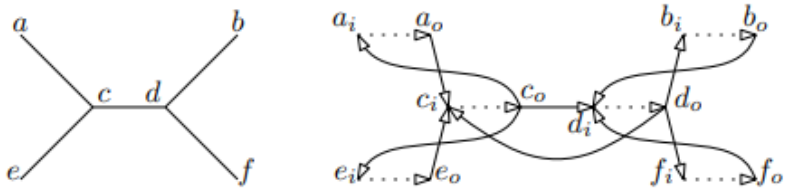
1. For each $x \in V$, add $V' = V' \cup \{x_i, x_o\}$ and $E' = E' \cup \{(x_i, x_o)\}$. x_o and x_i are called *out-node* and *in-node* of x respectively. Each (x_i, x_o) is called an internal edges.

2. For each arc $(u, v) \in E$, $E' = E' \cup \{(u_o, v_i)\}$. These arc are called external edges.

3. Compute a general cut of the graph G'. Let the cut edges be C'. C' may contain both internal and external edges. The external edges correspond to an edge in G and together they form a cut of G.

4. Let P be the node separator of graph G corresponding to the computed cut of G'. For each internal edge, include the corresponding node of G in P.For each external edge, include the endpoint which is in the larger side of G.

An example of conversion from G to G' is given in figure 31.



(a) Input graph (b) Expanded graph

Figure 31: Transforming a undirected graph G to directed graph G' for finding a node separator.

7.3 Tree Decomposition Using Nested Dissection Method

7.3.1 Overview of the Nested Dissection Method

A summary of the tree decomposition algorithm utilizing the nested dissection technique is shown in Figure 32. Each of the stages in the flowchart are explained in the following subsections.

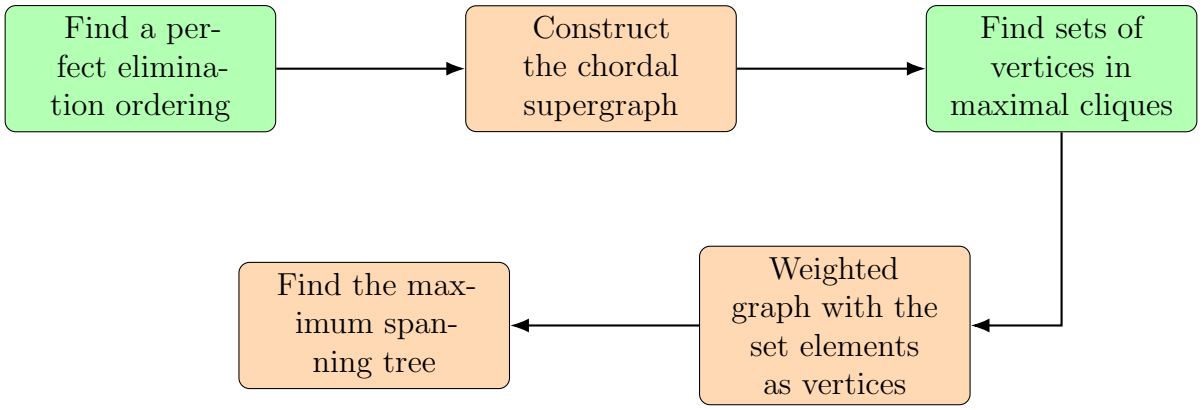


Figure 32: Flowchart of the tree decomposition algorithm using nested dissection method

7.3.2 Finding A Perfect Elimination Ordering

For this task the input graph G undergoes a process of recursive decomposition until G is reduced to isolated nodes. The reassembly is achieved by combining the orderings from the recursive calls, followed by the balanced separator node. Algorithm 7 describes the steps formally.

Algorithm 7 FindOrdering

Require: Graph G
Ensure: Perfect elimination ordering of G
1: **if** G has only one node **then**
2: **return** list containing the single node of G
3: **end if**
4: Identify a small, balanced node separator in G
5: Divide G into two subgraphs, G_1 and G_2, separated by the node separator
6: recursion_list_1 $\leftarrow$ FindOrdering(G_1)
7: recursion_list_2 $\leftarrow$ FindOrdering(G_2)
8: **return** Concatenation of recursion_list_1, recursion_list_2, and the node separator

In Figure 33, an example reassembly is shown. The recursive calls on the right returns a, e, d, c, b and g, j, i, h. The parent graph then merges these two perfect orderings by concatenating them and appending the node separator f at the end.

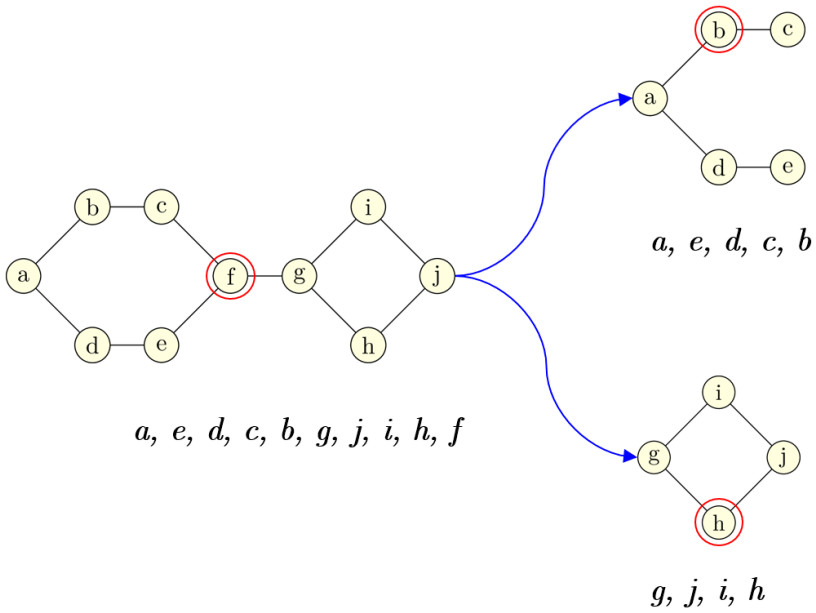


Figure 33: Merging the ordering from recursive calls

7.3.3 Constructing Chordal Supergraph

Given an elimination ordering E of a graph G, the chordal supergraph is constructed as follows: a copy of G is made and each vertex in E is considered in sequence. Edges are introduced between non-adjacent neighbors of the vertex in both graphs. Once the processing for a vertex is complete, the vertex is removed from the copied graph. Algorithm 8 describes the overall process formally.

Algorithm 8 ConstructChordalSupergraph

Require: Graph G, Elimination ordering E
Ensure: Chordal supergraph of G

```
Initialize G_workspace to a copy of G
Initialize G_final to a copy of G
for each vertex v in E, traversing from left to right do
    for each pair of neighbors u, w of v in G_workspace do
        if there is no edge between u and w then
            Add edge (u, w) to G_workspace
            Add edge (u, w) to G_final
        end if
    end for
    Remove vertex v from G_workspace
end for
return G_final
```

In Figure 34, the construction process is shown focusing on the vertex a. At first, edges

between non-adjacent neighbors of the vertex a are introduced in both the original and the duplicated graph. Subsequently, a is removed from the duplicated graph, and the process continues with the next vertex e defined by the sequence in the elimination ordering E.

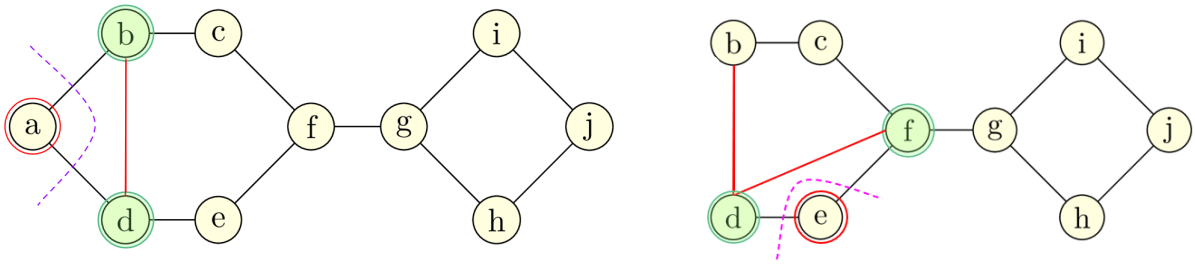


Figure 34: Construction process of the chordal supergraph focused on vertex a

When the construction process is completed, the resulting graph is shown in Figure 35. Newly added edges are colored red.

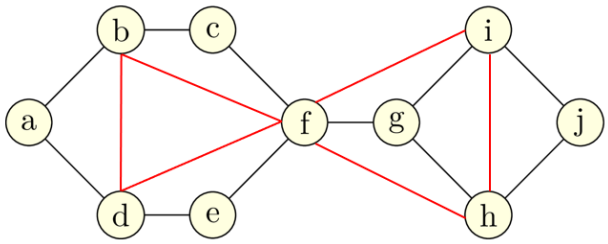


Figure 35: G_{final} after Algorithm 8 terminates

7.3.4 Finding Maximal Cliques

Algorithm 9 FindMaximalCliques

Require: Chordal graph G, Elimination ordering E
Ensure: Set of all maximal cliques in G
1: Initialize an empty set C to store cliques
2: **for** each vertex v in the elimination ordering E **do**
3: Find all neighbors N_v of v that are to its right in E
4: Form a clique K with v and its neighbors: $K = \{v\} \cup N_v$
5: Add the clique K to the set C
6: Remove v from the graph G
7: **end for**
8: **return** C

In order to find all the maximal cliques in a chordal graph G, Algorithm 9 starts with an empty set of cliques and considers the elimination ordering E found earlier. It iterates from

each vertex in the ordering E, constructs cliques with the vertex and its rightward positioned neighbors in the ordering, and adds these cliques to the set. Removal of the vertex at the end ensures that only rightward neighbors for the subsequent vertices are processed.

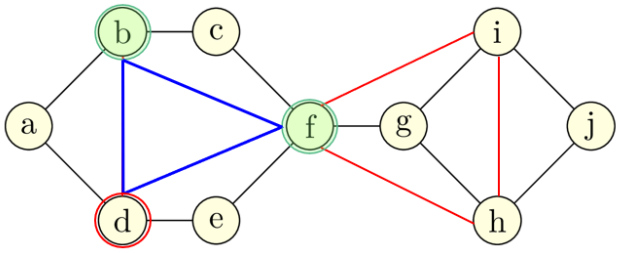


Figure 36: Maximal clique bdf considering the vertex d

Figure 36 illustrates the process of finding the maximal clique when vertex d is considered. The clique is formed by taking vertices b, d and f. Although vertices a and e are neighbors of d, those are not considered in the clique as their positions occur to the left of the vertex d in the elimination ordering E. The final set of cliques will be $\{abd, edf, dbf, cbf, bf, ghif, jih, ihf, hf, f\}$ when Algorithm 9 terminates.

7.4 Constructing Weighted Intermediate Graph

After identifying the set of cliques, a new intermediate graph G is constructed, where each clique from the set is represented as a vertex. Edges are introduced between vertices that share common elements, with the edge weight corresponding to the number of shared elements.

Algorithm 10 ConstructWeightedGraph

Require: Set of maximal cliques S
Ensure: Weighted graph G
Initialize G as an empty graph
for each clique C in S **do**
    Add a new vertex v to G representing the clique C
end for
for each pair of vertices u, v in G **do**
    $W \leftarrow$ the number of overlapping characters between the cliques represented by u and v
    if $W > 0$ **then**
        Add an edge (u, v) to G with weight W
    end if
end for
return G

For instance, we show an example in Figure 37. Between two vertices labeled abd and dbf, which share the elements d and b, an edge is established with a weight of 2 to denote these two

common elements. This process is similarly applied to all vertex pairs, which constructs the intermediate weighted graph. We only show the edges corresponding to vertices *abd* and *jih*, the edges corresponding to other vertices are implied by dashed lines.

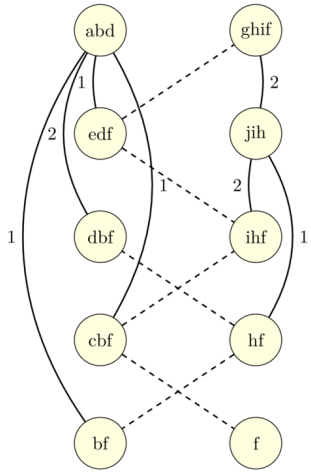


Figure 37: Intermediate graph

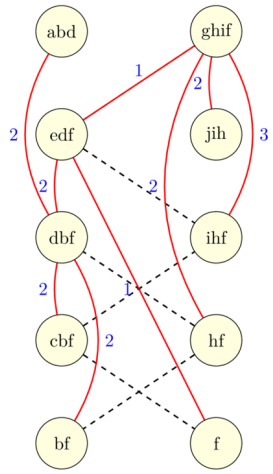


Figure 38: MST highlighted in red

7.5 Finding Maximum Spanning Tree

A Maximum Spanning Tree (MST) is a spanning tree of a weighted graph having maximum weight. We use the Kruskal's algorithm to find the MST in the weighted graph shown in 38. Algorithm 11 describes the overall steps formally.

Algorithm 11 Kruskal's Algorithm for Maximum Weight Spanning Tree

Require: Weighted connected graph $G(V, E)$
Ensure: Maximum weight spanning tree T
1: Sort the edges of G into decreasing order by weight
2: Initialize T as an empty set: $T \leftarrow \emptyset$
3: Add the first edge to T
4: Initialize n as the number of vertices in G
5: **repeat**
6: Consider the next edge in the sorted list
7: **if** adding the edge to T does not form a cycle **then**
8: Add the edge to T
9: **end if**
10: **if** there are no remaining edges **then**
11: Report G as disconnected and exit
12: **end if**
13: **until** T has $n - 1$ edges
14: **return** T

The MST in Figure 38 is redrawn in Figure 39 by taking vertex *edf* as the root of the tree. While this reconfiguration is not a prerequisite for the tree decomposition process, we do this to enhance the visual representation of the graph in order to render it with a more tree-like structure. The tree width for this tree decomposition is computed as follows: go through all the nodes and find the one with the maximum number of elements. In this case, this is the node labeled $ghif$. The tree width is computed by subtracting 1 from the maximum number of elements, in this case: width of the tree decomposition is $4 - 1 = 3$.

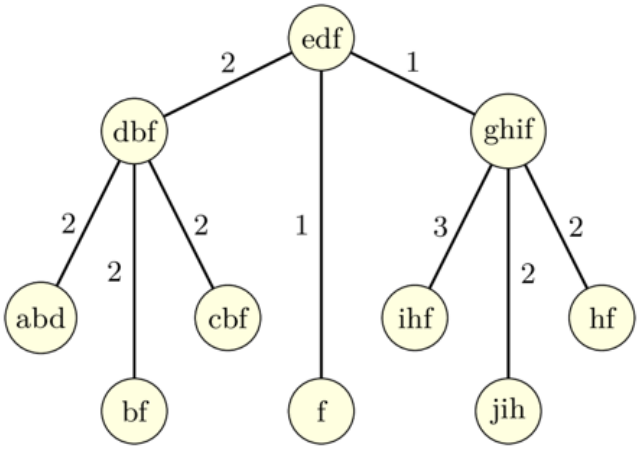


Figure 39: Redrawn maximum spanning tree

7.6 Results for the Flow Cutter Algorithm

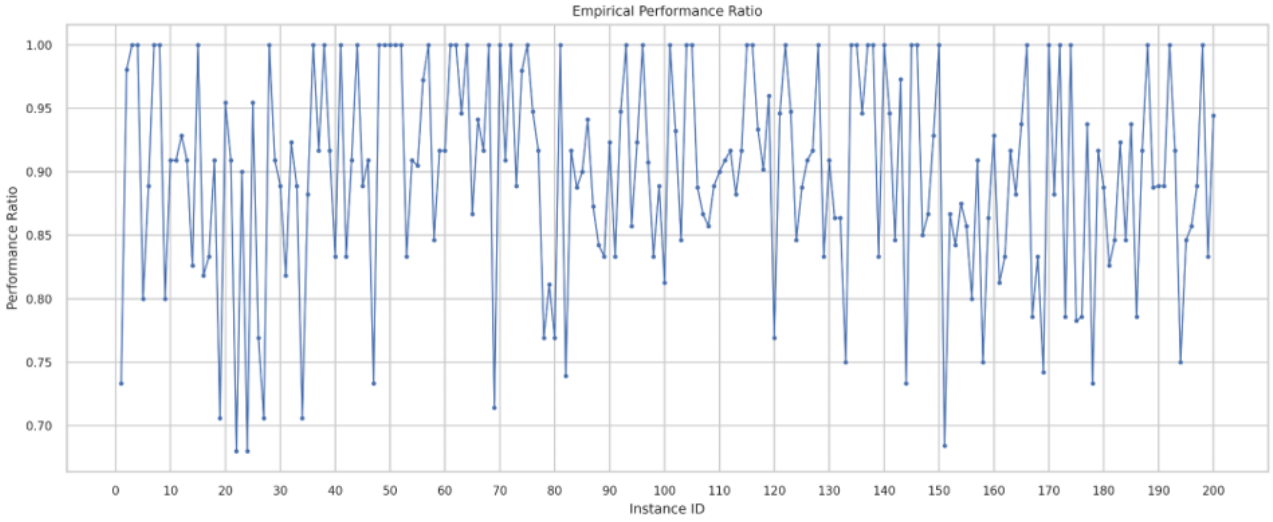


Figure 40: Performance ratio against instance IDs

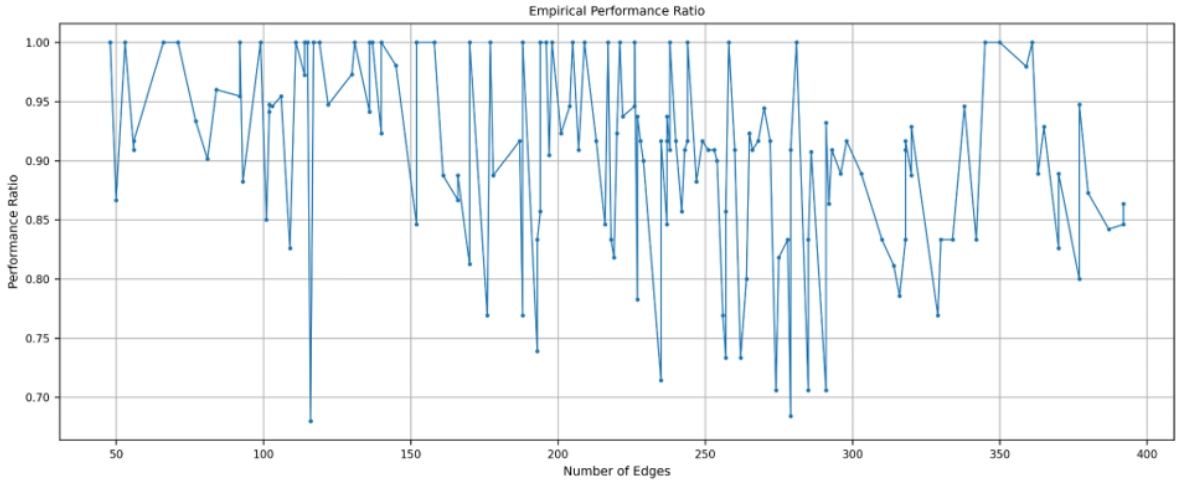


Figure 41: Performance ratio against number of edges

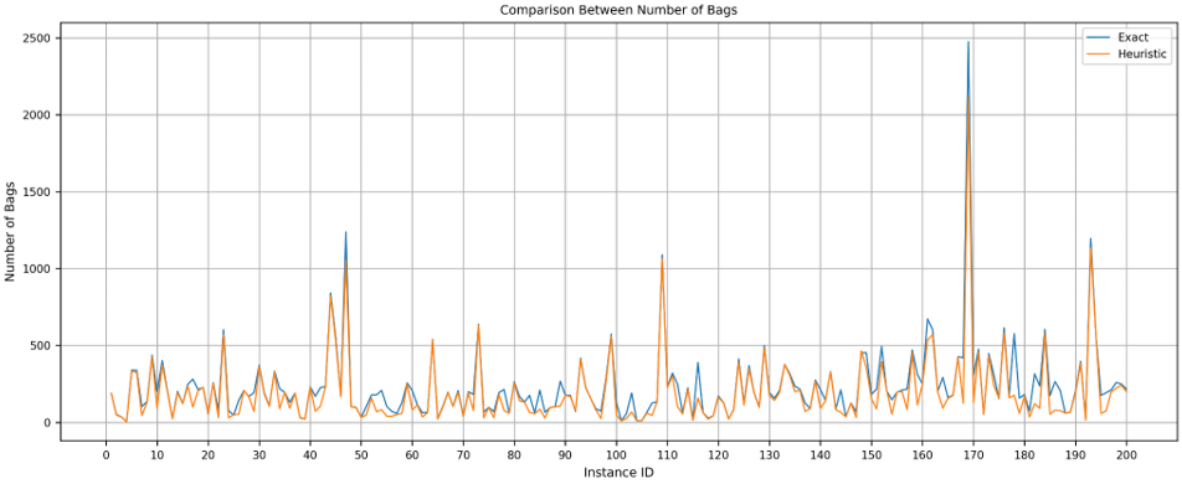


Figure 42: Number of bags between exact and heuristic solutions

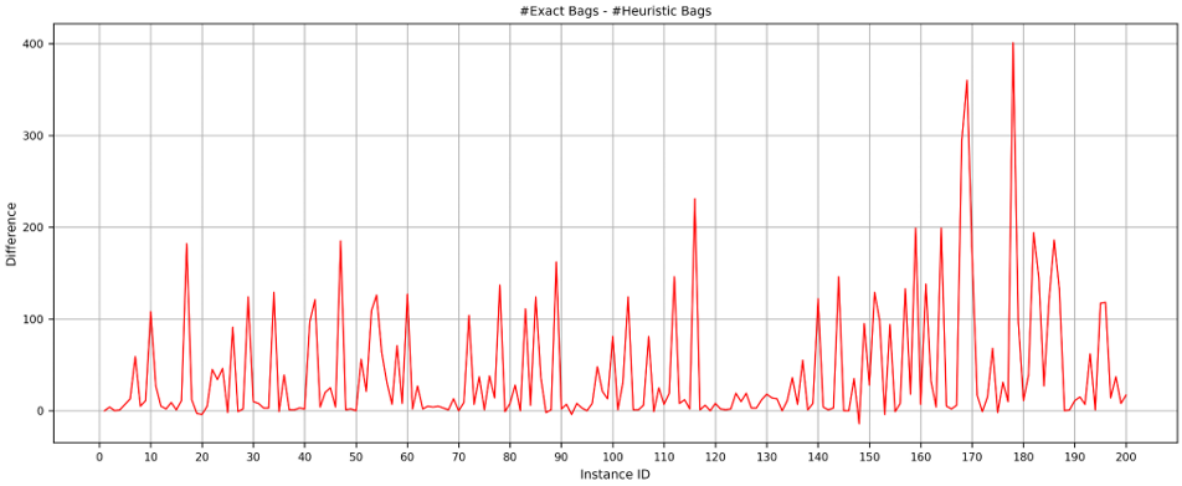


Figure 43: Difference of the number of bags between exact and heuristic solutions

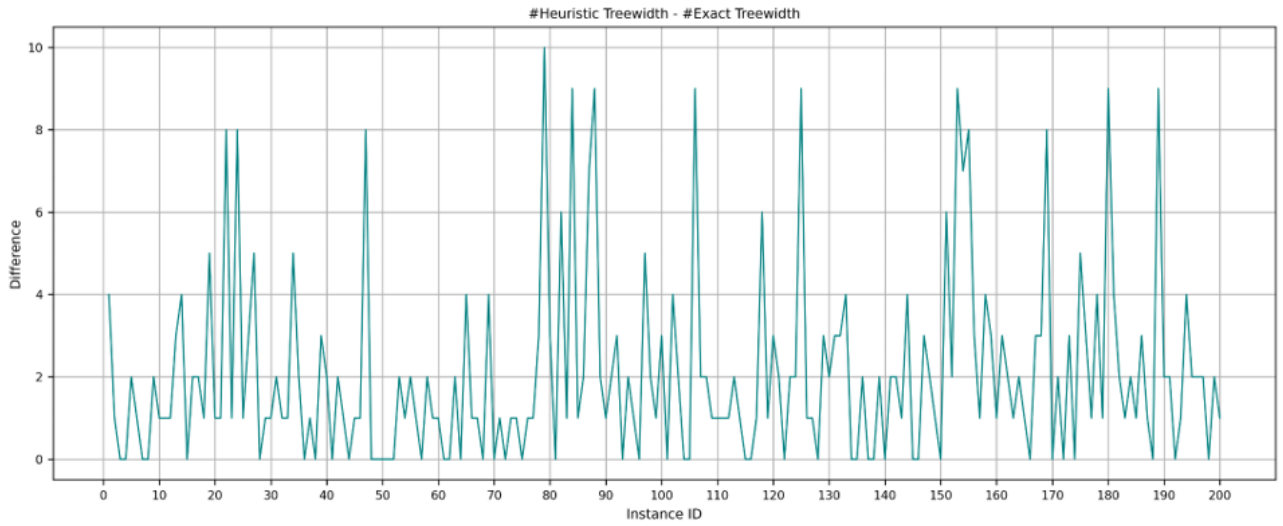


Figure 44: Difference of the tree widths between exact and heuristic solutions

Graphs were created by calculating a performance ratio: the tree width obtained from the heuristic algorithm was divided by the exact tree width of the solutions. This ratio was then plotted against various parameters such as instance IDs, number of edges, and number of bags. We also plotted the difference in the number of bags relative to the instance IDs. According to the challenge specification [17], The instances had been organized by increasing complexity, with later instances expected to be more challenging. This increment in difficulty was supported by the graphs, which showed a slightly upward trend in the difference of tree widths between the exact solutions and the heuristic outputs for subsequent instances.

8 Future Directions

1. Explore runtime vs Density of input graphs
2. Explore possible relationships between Chordal Supergraphs 7.1 and Minimal Chordal Completion 4.2.1
3. Explore performance ratio for the heuristic algorithm against density of the input graphs.

References

[1] H. Dell, C. Komusiewicz, N. Talmon, and M. Weller, "The PACE 2017 Parameterized Algorithms and Computational Experiments Challenge: The Second Iteration," in *12th International Symposium on Parameterized and Exact Computation (IPEC 2017)* (D. Lokshtanov and N. Nishimura, eds.), vol. 89 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 30:1–30:12, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

[2] H. L. Bodlaender, "A linear-time algorithm for finding tree-decompositions of small treewidth," *SIAM Journal on Computing*, vol. 25, no. 6, pp. 1305–1317, 1996.

[3] TCS Meiji, "Exact treewidth computation pid acp." GitHub repository, 2023.

[4] H. Tamaki, "Positive-instance driven dynamic programming for treewidth," 2018.

[5] M. Bannach, S. Berndt, and T. Ehlers, "Jdrasil: A modular library for computing tree decompositions," in *The Sea*, 2017.

[6] F. V. Fomin, I. Todinca, and Y. Villanger, "Large induced subgraphs via triangulations and CMSO," *CoRR*, vol. abs/1309.1559, 2013.

[7] M. Belbasi and M. Fürer, "An improvement of reed's treewidth approximation," *CoRR*, vol. abs/2010.03105, 2020.

[8] T. Korhonen and D. Lokshtanov, "An improved parameterized algorithm for treewidth," 2023.

[9] M. Hamann and B. Strasser, "Graph bisection with pareto-optimization," 2017.

[10] H. L. Bodlaender, "Treewidth: characterizations, applications, and computations," in *Proceedings of the 32nd International Conference on Graph-Theoretic Concepts in Computer Science*, WG'06, (Berlin, Heidelberg), p. 1–14, Springer-Verlag, 2006.

[11] V. Gogate and R. Dechter, "A complete anytime algorithm for treewidth," 2012.

[12] S. Arnborg and A. Proskurowski, "Linear time algorithms for np-hard problems restricted to partial k-trees," *Discrete Applied Mathematics*, vol. 23, no. 1, pp. 11–24, 1989.

[13] S. Arnborg, D. G. Corneil, and A. Proskurowski, "Complexity of finding embeddings in a k-tree," *SIAM Journal on Matrix Analysis and Applications*, vol. 8, no. 2, pp. 277–284, 1987.

[14] M. Yannakakis, "Computing the minimum fill-in is np-complete," *SIAM Journal on Algebraic and Discrete Methods*, vol. 2, no. 1, pp. 77–79, 1981.

[15] H. L. Bodlaender and A. M. Koster, "Safe separators for treewidth," *Discrete Mathematics*, vol. 306, no. 3, pp. 337–350, 2006. Minimal Separation and Minimal Triangulation.

[16] M. Hamann and B. Strasser, 2017. https://github.com/kit-algo/flow-cutter-pace17.

[17] "PACE 2017 Track A: Treewidth." https://pacechallenge.wordpress.com/pace-2017/track-a-treewidth/, 2017. Accessed: 2024-03-09.