Parameterized Algorithms for Graph Problems

A Thesis

submitted to
Indian Institute of Science Education and Research Pune
in partial fulfillment of the requirements for the
BS-MS Dual Degree Programme

by

Vaikunt Mallya

Indian Institute of Science Education and Research Pune
Dr. Homi Bhabha Road,
Pashan, Pune 411008, INDIA.

October, 2021

Supervisor: Dr. Soumen Maity

Certificate

This is to certify that this dissertation entitled Parameterized Algorithms for Graph Problems towards the partial fulfilment of the BS-MS dual degree programme at the Indian Institute of Science Education and Research, Pune represents study/work carried out by Vaikunt Mallya at Indian Institute of Science Education and Research under the supervision of Dr. Soumen Maity, Associate Professor, Department of Mathematics , during the academic year 2020-2021.

Dr. Soumen Maity

Committee:

Dr. Soumen Maity

Dr. Vivek Mallick

This thesis is dedicated to IISER Pune and
all the experiences in my time there.

Declaration

I hereby declare that the matter embodied in the report entitled Parameterized Algorithms for Graph Problems are the results of the work carried out by me at the Department of Mathematics, Indian Institute of Science Education and Research, Pune, under the supervision of Dr. Soumen Maity and the same has not been submitted elsewhere for any other degree.

Vaikunt Mallya

Acknowledgments

First and Foremost I would like to express my deepest gratitude for my project supervisor Dr. Soumen Maity for providing me this opportunity and the for incredible amount of patience, support and encouragement he have given me during the course of this project without which the completion of this project would have been impossible.

I would especially like to show my appreciation to the various faculty members of the Department of Mathematics, IISER Pune who always entertained my doubts and worries and gave me inspiration to pursue my interest in a career of mathematics.

I would also like to thank IISER Pune for providing me with many opportunities to gain experience in many fields of science and granting me very fulfilling college education.

I am grateful for the constant support and motivation my parents have given me in my choice of education and career path.

I would lastly like to thank my friends Vyshnav, Raja D, Ayushman, Babu, Lagachu, Saumil, Shaun, Haris, JJ, Karan, Soman, Bala, Bhole, Rout and many more for helping me keep my sanity intact over the past year of isolation. A special thanks to my college seniors Shashank, Chandra, Chary and Vishak for helping me with many doubts during the years of course work. No claims to the originality of the content in this paper is made whatsoever and the credit is given where due in the references. If any reference is missing kindly contact me and it will be rectified.

Abstract

The fixed parameter approach to a problem is a technique of designing an algorithm to solve combinatorially hard problems. A parameterized problem has an input instance x as well as a parameter k which is sufficiently small compared to the input size of the instance. A parameterized problem is FPT (fixed parameter tractable) if there exists an algorithm which correctly decides whether the input instance of the given problem is a yes or no instance in $f(k).|x|^{O(1)}$ time where f is some computable function. Similarly, if the algorithm requires $f(k).|x|^{g(k)}$ time then it is called XP (here f and g are computable functions). In this project, we study paramaterized complexity, kernelization techniques, branching algorithms and treewidth.

Contents

Chapter 1

Introduction

In this thesis we will study various methods of parameterizing a problem and see how these parameters affect the computability or efficiency of the solution.

1.1 Parameterization and Complexity

Computation problems are classified on the basis of their complexity. The complexity of a problem is defined to be the run time of the algorithm which solves the problem. An "efficient" algorithm is one which runs in time polynomial in the size of the input, to output a solution. The problems which can be solved in polynomial time are considered to be easy whereas those problems which require super-polynomial time algorithms are considered to be difficult. These computationally hard problems are classified as NP-hard and are not known to have a polynomial time solution, nor has the existence of such a solution been proven.

Some approaches to such problems are approximation and parameterization. Algorithms which use approximation are those that run in polynomial time to output a near optimal solution. We try to obtain a polynomial time solution by relaxing the constraint on optimality of the solution. To verify if our approximate solution is close to optimal we must compare it to the optimal solution, which is very difficult to compute.

For the approach of parameterization, we know that not all instances of an NP-hard problem are equally hard to solve, rather their difficulty or hardness depends on the structure of the given instance. In a parameterized problem we have for some input instance I and parameter k that is chosen to be a sufficiently small on the input size of the instance. The art of parameterization involves choosing the best parameter to as to restrict the combinatorial explosion and to have a very efficient algorithm.

Consider some size-n instance of a fixed-parameter tractable problem, with some parameter k, this can be solved in $f(k)\cdot p(n)$ time, where function f depends only on the parameter k and $p(n)$ is just a polynomial in n, establishing the FPT of a problem confines the combinatorial explosion to the parameter. Formal definitions of FPT and related topics are provided in the following chapter.

1.2 Goal of this thesis

The goal of this thesis is to survey various techniques of fixed parameter tractable problems. In particular we will discuss:

- Kernelization methods
- Bounded Search Tree
- Treewidth

These topics will be covered at various chapters of the thesis and some NP-hard problem are used as an example for the various parameterization techniques mentioned above. The goal is to gain a better understanding of these topics.

1.3 Original Contribution

This dissertation is primarily a literature review to learn and understand few subtopics of parameterized algorithms. Through this work, it is expected that the reader can get quick

insight into core topics of the subject without the process of going through standard textbook material and to provide quick access to some standard problems with original examples and illustrations. Across the various chapters the basics of the subject have been explained with a focus on a few chapters from the book *Parameterized Algorithms* by Cygan(et al.)[1] and papers based on the same or similar topics by various authors. The various chapters have the core source material based on the content in these references while the examples and illustrations are contributed by the author of this paper.

Chapter 2

Preliminaries

In this chapter, we cover some basic theorems and definitions. Any other definitions required for a specific topic will be explained in respective sections

2.1 Graphs

- A graph G is represented as $G = (V, E)$, where V and E are the vertex and edge set of that graph respectively. Unless explicitly stated otherwise, we assume G to be a simple graph which is not directed.
- If u, v are two vertices of G, then uv represents an edge connecting the two vertices and the existence of this edge makes u and v adjacent.
- The neighbourhood of a vertex v is defined as $N(v) = \{u : uv \in E(G)\}$ and the closed neighbourhood of v is defined as $N[v] = N(v) \cup \{v\}$.
- A vertex v is isolated if it does not have an edge with any other vertex u of the graph.
- The independent vertex set is a subset I of $V(G)$, if every pair of vertices in G are non-adjacent.
- A clique $C \subseteq V(G)$ is a set of vertices such that every pair of vertices in C is adjacent.
- the degree $\delta(v)$ of a vertex v is said to be the number of edges incident on it.

- Matching

  A matching in a graph $G = (V, E)$ is a collection of pairwise disjoint edges. A vertex cover on a graph $G = (V, E)$ is a set $C \subseteq V(G)$ of vertices such that every edge of G is incident to atleast one vertex of C.

  Example 1. Consider the following bipartite graph G show in figure 2.1

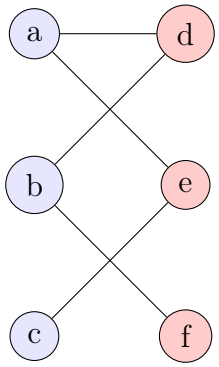


Figure 2.1: Example of Matching

  Note that $M = \{(a, d), (b, f), (c, e)\}$ is a matching of size 3 and this is a maximum matching.
  Clearly $C = (d, e, f)$ is a vertex cover of minimum size. Observe that the size of maximum matching in G is equal to the size of minimum vertex cover.

- Konig's Theorem

  The size of a maximum matching is equal to the size of a minimum vertex cover in every undirected bipartite graph.

- Halls's Theorem

  Given an undirected bipartite graph G with Bipartition (V_1, V_2), G has a matching saturating V_1 if and only if for $X \subseteq V_1$ we have that , $|X| \leq |N(X)|$

- Hopcroft-Karp Theorem

  Given an undirected bipartite graph G with bipartition (V_1, V_2) on n vertices and m edges, the maximum matching and minimum vertex cover of G can be obtained in time $\mathcal{O}(m\sqrt{n})$. In this time $\mathcal{O}(m\sqrt{n})$ we either obtain a matching saturating V_1 or a inclusion-wise minimal set $X \subseteq V_1$ such that $N|X| \leq |X|$

2.2 Parameterized Problems

Definition 2.2.1. A Language $L \subseteq \Sigma^* \times \mathbb{N}$ is called a parameterized problem, where Σ is a finite fixed alphabet. k is called the parameter for a given instance $(x, k) \in \Sigma^* \times \mathbb{N}$ where Σ^* is the set of strings for that language.

Definition 2.2.2. For an instance (x, k), its size is defined as $|x| + k$

Definition 2.2.3. If there exists an algorithm $\mathscr{A}$, a computable function $f : \mathbb{N} \to \mathbb{N}$ and a constant c(i.e c $= \mathcal{O}(1)$) such that $\mathscr{A}$ correctly decides for some given $(x, k) \in \Sigma^* \times \mathbb{N}$ if $(x, k) \in L$ within a time bound of $f(k)|(x, k)^c|$ for some parametrized problem $L \subseteq \Sigma^* \times \mathbb{N}$. Then the algorithm $\mathscr{A}$ is called a fixed parameter algorithm, and the problem L is called fixed parameter tractable or FPT.

Definition 2.2.4. Given an algorithm $\mathscr{A}$ and 2 computable functions $f, g : \mathbb{N} \to \mathbb{N}$ such that $\mathscr{A}$ correctly decides for some given $(x, k) \in \Sigma^* \times \mathbb{N}$ if $(x, k) \in L$ within a time bound of $f(k)|(x, k)^{g(k)}|$ for some parametrized problem $L \subseteq \Sigma^* \times \mathbb{N}$. Then the problem L is called Slice-wise Polynomial or XP.

Chapter 3

Kernelization Techniques

The process of efficiently solving the easier parts of a problem, so as to reduce it to its computationally difficult core structure is known as Preprocessing or Kernelization. These algorithms try to reduce a given problem instance to equivalent smaller instance if possible within polynomial time. In this chapter we will look at definitions of kernelization and how reduction rules are used in certain problems to obtain their kernels.
Original contribution includes the restructuring of topics, proofs to certain lemmas and theorems, and adding of illustrations to improve the readability and understanding of the topics.

3.1 Definitions

Definition 3.1.1. A function $\gamma : \Sigma^* \times \mathbb{N} \to \Sigma^* \times \mathbb{N}$ that takes an instance (J, k) of a parameterized problem $\mathscr{P}$ and maps it to an equivalent instance (J', k') of $\mathscr{P}$ such that γ is computable in polynomial time in $|J|$ and k is called a data reduction rule.

Definition 3.1.2. If $(J, k) \in \mathscr{P}$ if and only if $(J', k') \in \mathscr{P}$ then instances (J, k) and (J', k') are said to be equivalent.
The property of the reduction rule γ which maps one instance to its equivalent is referred to as its safeness.

Definition 3.1.3. Given a reduction algorithm $\mathscr{A}$, its output size is a function $size_{\mathscr{A}}$:

$\mathbb{N} \to \mathbb{N}$ defined as $size_{\mathscr{A}}(k) = sup\{|J'| + k' : (J', k') = \mathscr{A}(J, k), J \in \Sigma^*\}$

Definition 3.1.4. Given a parameterized problem $\mathscr{P}$ and a computable function $q : \mathbb{N} \to \mathbb{N}$. An algortithm $\mathscr{A}$ which takes an instance (J, k) of $\mathscr{P}$ and works in polynomial time to output an equivalent instance (J', k') of $\mathscr{P}$ such that $size_{\mathscr{A}}(k) \leq q(k)$ is called a kernelization algorithm.
If q is a linear/polynomial function in $|J'| + k' \leq q(k)$ then accordingly $\mathscr{P}$ admits a linear/polynomial kernel.

Lemma 3.1.1. If some parameterized problem $\mathscr{P}$ is FPT then it admits a kernelization algorithm.

3.2 Kernel for the VERTEX COVER problem

A subset $S \subseteq V(G)$ is called the vertex cover of $G = (V, E)$ if every edge of G contains at least one endpoint in S. In Figure 3.1 we have $S = (v_4, v_5, v_6, v_{10})$ to be one vertex cover of the graph.

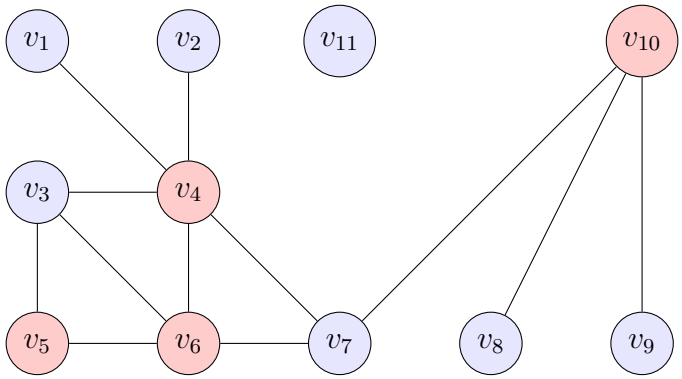


Figure 3.1: Example of Vertex Cover

Given a graph G and an integer k, the VERTEX COVER problem asks if G has a vertex cover S such that $|S| \leq k$. We will now discuss reduction rules to tackle an instance the parameterized vertex cover for some given instance (G, k).

Reduction 3.2.1. An isolated vertex v does not cover any edge and thus can be deleted. This gives us the new instance as $(G - v, k)$

Reduction 3.2.2. If there is a vertex v such that $\deg(v) \geq (k+1)$, then this vertex must be included in every vertex cover of size at most k, thus we can delete the vertex and its edges from G and decrease the parameter k by 1. This gives us the new instance as $(G-v, k-1)$.

Example 2. Suppose input to our kernelisation algorithm is the graph G, given in Figure 3.1 and parameter $k = 4$. Then after application of Reduction 3.2.1 we have our new instance as the graph shown in Figure 3.2 and $k = 4$.

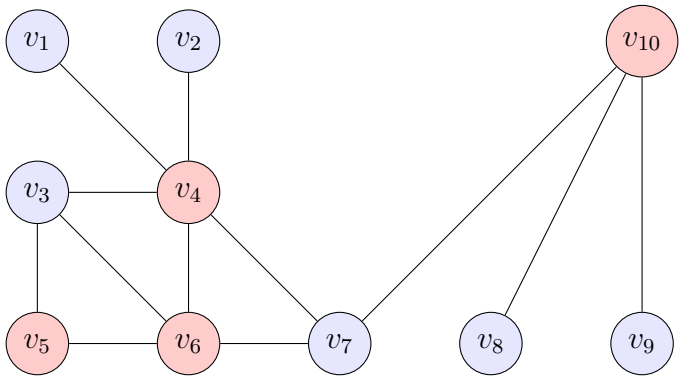


Figure 3.2: New instance after Reduction 3.2.1

After application of Reduction 3.2.2 we have our new instance which is shown in Figure 3.3 below and $k = 3$.

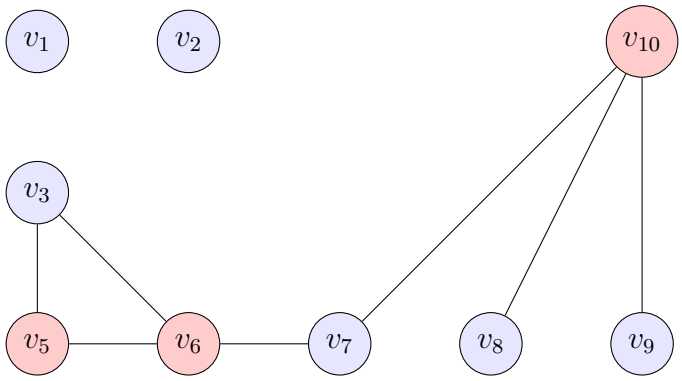


Figure 3.3: New instance after Reduction 3.2.2

After applying Reduction rule 3.2.1 once more, the new instance is the graph shown in Figure 3.4 and $k = 3$.

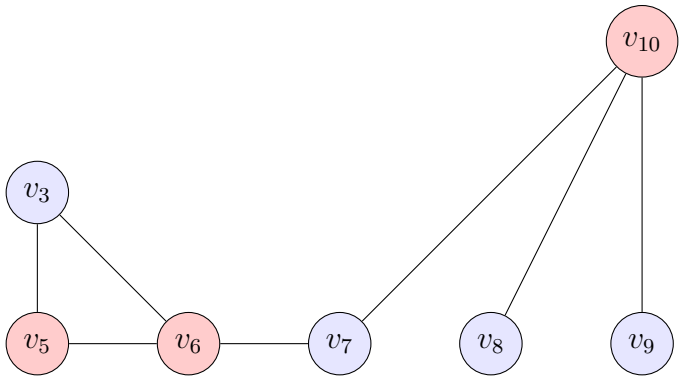


Figure 3.4: Kernel

Now, we see that none of the reduction rules are applicable and this is our kernel.

Remark 3.2.1. Note if a graph is such that it has a maximum degree δ, then a set consisting of k vertices can cover a maximum of $k\delta$ edges.

Lemma 3.2.3. *If our instance (G, k) is a yes-instance and reduction rules 3.2.1 and 3.2.2 are not applicable, then we have that $|V(G)| \leq k^2 + k$ and $|E(G)| \leq k^2$.*

Proof. As rule 3.2.1 is not applicable we have that given any vertex cover S of G, every vertex of $G - S$ must be adjacent to some vertex from S. As rule 3.2.2 is also not applicable we have that the maximum degree δ of the graph is such that $\delta \leq k$. Thus we have $|V(G - S)| \leq k|S|$ and as such $|V(G)| \leq (k + 1)|S|$. Since it is a yes-instance $\exists$ a vertex cover S such that $|S| \leq k$. Thus we have $|V(G)| \leq (k + 1)k$. As each vertex can only cover a maximum of k edges, if G has more than k^2 edges, it becomes a no instance.

Reduction 3.2.4. Assume for instance (G, k) we have that reduction rules 3.2.1 and 3.2.2 are not applicable. Then if $k < 0$ and $V(G) > (k^2 + k)$ or $E(G) > k^2$ then we conclude it is a no instance.

Theorem 3.2.5. *The* Vertex Cover *problem admits a kernel with $\mathcal{O}(k^2)$ vertices and $\mathcal{O}(k^2)$ edges.*

3.3 Feedback Arc Set in Tournaments

A tournament $\mathcal{T} = (V, E)$ is complete directed graph, that is, for every pair of vertices $(a, b) \in VXV$ either (a, b) is a directed edge or (b, a) is a directed edge in the tournament.

Definition 3.3.1. A feedback arc set A is a set of directed edges of a directed graph G such that $G - A$ is a directed acyclic graph.

Example 3. Consider the tournament $\mathcal{T}$ as shown in figure 3.5. It has a feedback arc set of size 1 and it consists of the arc (b, c).

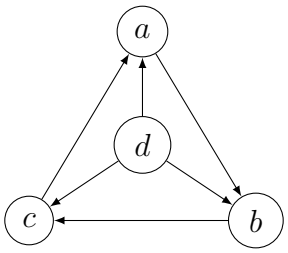


Figure 3.5: Tournament $\mathcal{T}$

□

Given a tournament $\mathcal{T}$ and a positive integer k, the problem is to find whether $\mathcal{T}$ contains a feedback arc set of size at most k.

Lemma 3.3.1. *A digraph G is acyclic if and only if there exists an ordering of its vertices such that for all directed edges (a, b) we have that $a < b$.*

Proof. Assume there exists a directed graph G with n vertices i.e $V(G) = \{a_1, a_2, ..., a_n\}$. If there exists ordering in these vertices such that for $i, j \in \mathbb{N}$ we have $a_i a_j \in E(G)) \implies i < j$ i.e $a_1 < a_2 < .....a_{n-1}, a_n$. Thus if this graph has a directed cycle then there would be and edge $a_n a_1 \in E(G)$ which would imply $a_n < a_1$ which is a contradiction. Thus If there is a ordering the digraph G must be acyclic.
Let $|V(G)| = n$ and let $V(G) = \{a_1, a_2, ..., a_n\}$ be the set of vertices for some DAG G_n. We have for $n = 1$, the topological ordering exists as there is only one vertex. For $n = 2$ we have only one edge and as the graph is DAG we can give an ordering $a_1 < a_2$ for the single edge $a_1 a_2$. Assuming for $n = k$ there exists an ordering such that $a_1 < a_2 < ..... < a_k$. Then for a graph with $n = k + 1$ we have a graph $G_{k+1} = G_k + a_{k+1}$. For graph G_k we have that since it is a DAG $a_i a_j \in E(G_k) \implies i < j$. Thus if a_{k+1} is added to G_k it must be added as a sink to maintain the ordering of G_k since the digraph is acyclic as if it is added as the source then edge $a_k a_{k+1}$ which is allowed by the ordering would make the graph cyclic.

Definition 3.3.2. Given digraph G and a subset $C \subseteq E(G)$, we define $G \star C$ to be the digraph obtained by reversing all edges in C i.e $Rev(C) = \{(v,u) : \forall (u,v) \in C\}$ with the edge set $E(G \star C) = (E(G) \cup Rev(C)) \setminus F$ and vertex set $V(G)$.

Remark 3.3.1. For digraph G given $C \subseteq E(G)$, if $G \star C$ is DAG we have that C is a feedback arc set of G.

Lemma 3.3.2. Given digraph G and $C \subseteq E(G)$. Then C is an inclusion-wise minimal feedback arc set of G if and only if C is an inclusion-wise minimal set of edges such that $G \star C$ is a DAG.

Theorem 3.3.3. Feedback arc set in tournaments admits a kernel with maximum size of $k^2 + 2k$

By the lemma 3.3.2, the maximum size of a feedback arc set for a tournament $\mathcal{T}$, is k if and only if it takes the reversing of a maximum of k edges, to make $\mathcal{T}$ an acyclic tournament.

Reduction 3.3.4. If at least $k+1$ triangles contain an edge e, then reverse e and reduce k by 1.

Reduction 3.3.5. If some vertex v does not belong to any triangle in $\mathcal{T}$, then delete v

Example 4. Suppose the input instance to our reduction algorithm is the tournament $\mathcal{T}$ shown in Figure 3.5 and $k = 1$.
Reduction rule 3.3.4 is applicable since the edge (b, c) is contained in 2 cycles; so we reverse (b, c) and reduce k by 1. The new instance is show in Figure 3.6 and k=0.

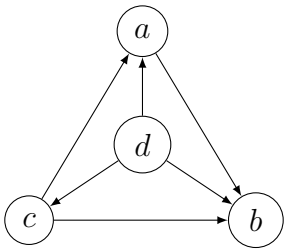


Figure 3.6: Tournament $\mathcal{T}$ with (b, c) reversed.

Reduction rule 3.3.5 is applicable as a is not contained in any triangle and we delete a. The new instance is shown in Figure 3.7 and $k = 1$.

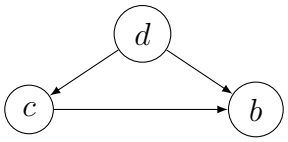


Figure 3.7: Instance after reduction 3.3.5

Reduction rule 3.3.4 is not applicable but Reduction rule 3.3.5 is applicable as b is not continued in any triangle. The new instance is shown in Figure 3.8 and $k = 0$.

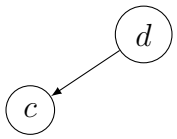


Figure 3.8: Instance after repeated reduction 3.3.5

Reduction rule 3.3.4 is applicable as c is not contained in any triangle and the new instance after reduction is just the isolated vertice d and $k = 0$. clearly, it is a yes-instance.

3.4 Edge Clique Cover

An edge clique cover is a set of cliques that cover all the edges.

Example 5. Consider the graph show in Figure 3.9. The edge clique cover in this graph is of size 4.

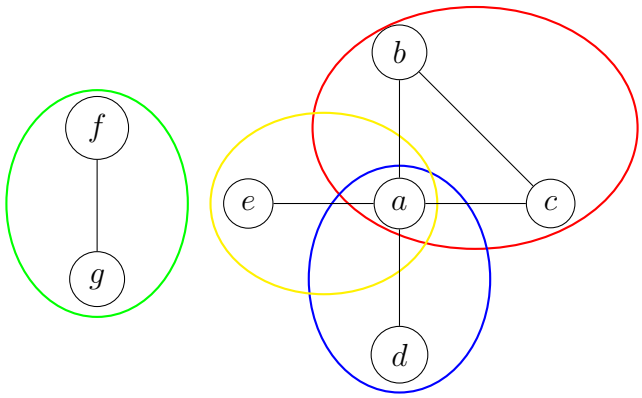


Figure 3.9: Example of edge clique cover

The cliques in the edge clique cover are $\{a, e\}, \{a, d\}, \{a, b, c\}$ and $\{f, g\}$.

□

Given a graph G and a non negative integer k, the goal is to decide whether there are at most k cliques that cover all the edges of G. To get a kernel for the input instance we apply the following reduction rules in the following order:

Reduction 3.4.1. Remove isolated vertices.

Reduction 3.4.2. Delete an isolated edge uv, and reduce k by 1. So the new instance is $(G - \{u, v\}, k - 1)$

Reduction 3.4.3. If $N[u] = N[v]$, for some uv, then delete v. This makes the new instance $(G - v, k)$

Example 6. Suppose the input instance of the kernelization algorithm is the graph shown in Figure 3.9 and $k = 4$. Reduction rule 3.4.1 is not applicable. After application of Reduction rule 3.4.2 we get the following graph as shown in Figure 3.10 and $k = 3$.

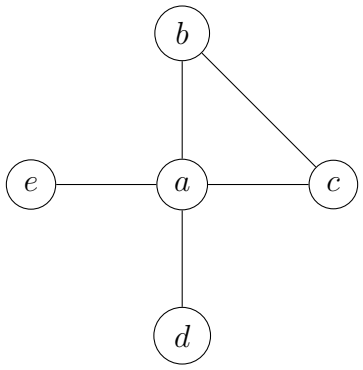


Figure 3.10: New instance after Reduction 3.4.2

Note that vertices b and c are adjacent and $N[b] = N[c]$. So we delete b. The new instance is shown in Figure 3.11 and $k = 3$.

Now we notice that none of the reduction rules are applicable and the graph shown in the Figure 3.11 is our final kernel.

□

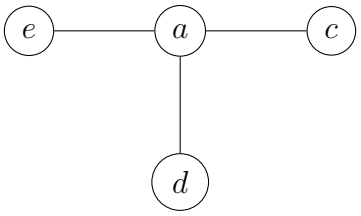


Figure 3.11: New instance after Reduction 3.4.3

Theorem 3.4.4. *For the problem of edge clique cover, there exists a kernel with maximum* 2^k *vertices.*

Proof. We claim that if G has a Edge Clique Cover of size k and no reduction rules are applicable, then G has at most 2^k vertices.

Suppose $C_1, C_2, ...C_k$ are the k cliques that cover G and assume G has $|V(G)| > 2^k$. Now we assign a vector y_v of length k to each vertex $v \in G$. The ith bit of y_v is set to 1 if and only if v is present in the $i$the clique,else it is 0

As there are only k positions each with only 2 possible values, we have that at max there can only be 2^k distinct vectors. As we assumed there to be more than 2^k vertices, there must be 2 vertices v_1, v_2 such that $v_1 \neq v_2$ and $y_{v_1} = y_{v_2}$. now ify_{v_1} and y_{v_2} are both zero vectors, then the respective vertices are isolated and Reduction 3.4.1 is applicable. If they are not zero vectors, then both vertices are in the same clique, are adjacent and have $N[v_1] = N[v_2]$, which means either reduction 3.4.2 or 3.4.3 is applicable. As such if G has more than 2^k vertices, the reduction rules are applicable, thus proving that there can be a maximum of 2^k vertices in G.

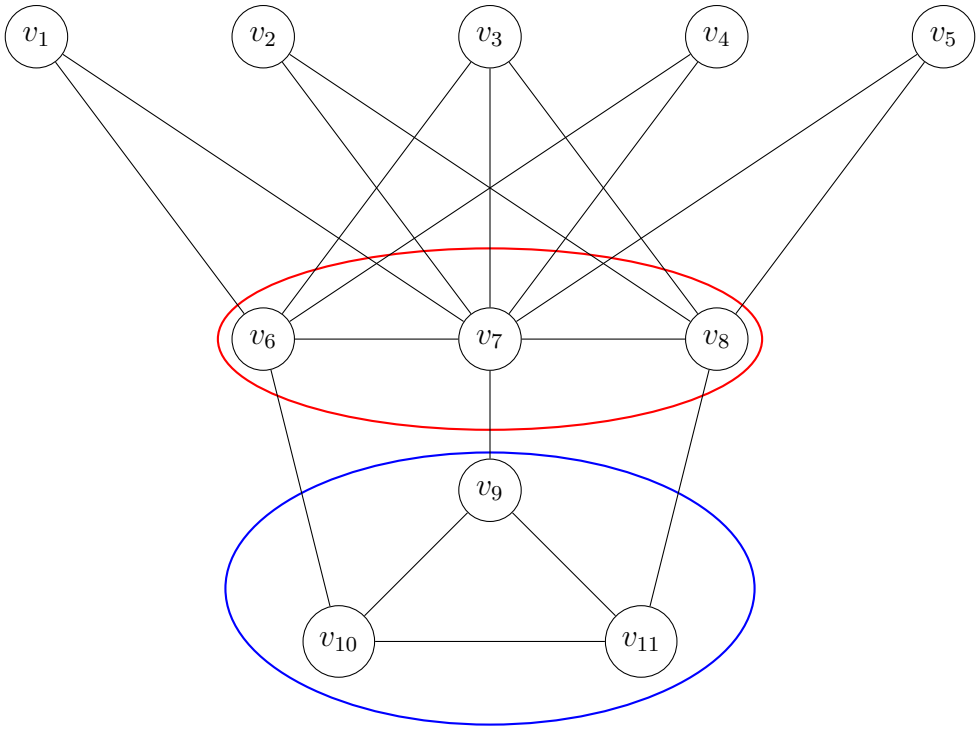


Figure 3.12: Example of Crown Decomposition

3.5 Crown Decomposition

Given a graph G the partion of the $V(G)$ into 3 parts C, H and R, such that

1. C is nonempty and independent.
2. H seperates C and R, i.e there are no edges between R and C.
3. if E' is the set of edges between C and H, then E' contains a matching of size $|H|$.

In the Figure 3.12, we have the crown decomposition of a graph G, where $C = \{v_1, v_2, v_3, v_4, v_5\}$ is an independent set. $H = \{v_6, v_7, v_8\}$, separates C from $R = \{v_9, v_{10}, v_1 1\}$. The matching $M = \{v_2v_8, v_3v_7, v_4v_6\}$ saturates all vertices of H.

Lemma 3.5.1 (Crown Lemma)**.** Let G be a graph with atleast $3k+1$ vertices and without isolated vertices. Then there exists a polynomial time algorithm which either

- Finds a matching of size $k+1$ in G.
- Finds a crown decomposition of G.

We can understand this through an example of the Vertex Cover problem.

Theorem 3.5.2. Vertex cover admits a kernel with at most $3k$ vertices.
Proof. Given some instance (G, k) of the Vertex Cover problem, all isolated vertices can be deleted by repeated use of rule 3.2.1, as such we can assume there are no isolated vertices in G. Now if $V(G) > 3k$, we can apply Lemma 3.5.1 to get either a matching of size $k+1$ or a crown decomposition of $V(G)$, i.e $V(G) = C \cup H \cup R$
If matching of size $k+1$ exists, then we can easily get a vertex cover of size $k+1$ by selecting one vertex from each edge in the matching which would imply we have a no instance.
In the case we have a decomposition, let $\mathcal{M}$ be the matching of H into C. Now we must select from $H \cup C$, at least $|\mathcal{M}| = |H|$ vertices in each cover. As H covers all edges of G incident on $H \cup C$ (since C is independent). Thus H must be contained in the minimum vertex cover of G, as such our instance (G, k) is reduced to $(G - H, k - |H|)$. Vertices of C are removed as they become isolated after removing H.
As $H \neq \emptyset$ by using crown lemma we can always reduce the graph when $|V(G)| > 3k$.

3.6 Sunflower Lemma

A sunflower with p petals is a collection of sets $S_1, S_2, ...S_p$ which have the same pairwise intersection, i.e $S_i \cap S_j = X \ \forall \ i \neq j$, X is called the core. the sets $S_i \setminus X$ are called petals and must be non empty, however X can be empty. In figure 3.13, the core center has colored yellow nodes and highlighted as the common intersection of the petals by a red circle. The petals are highlighted by blue loops which encase nodes of various colors.

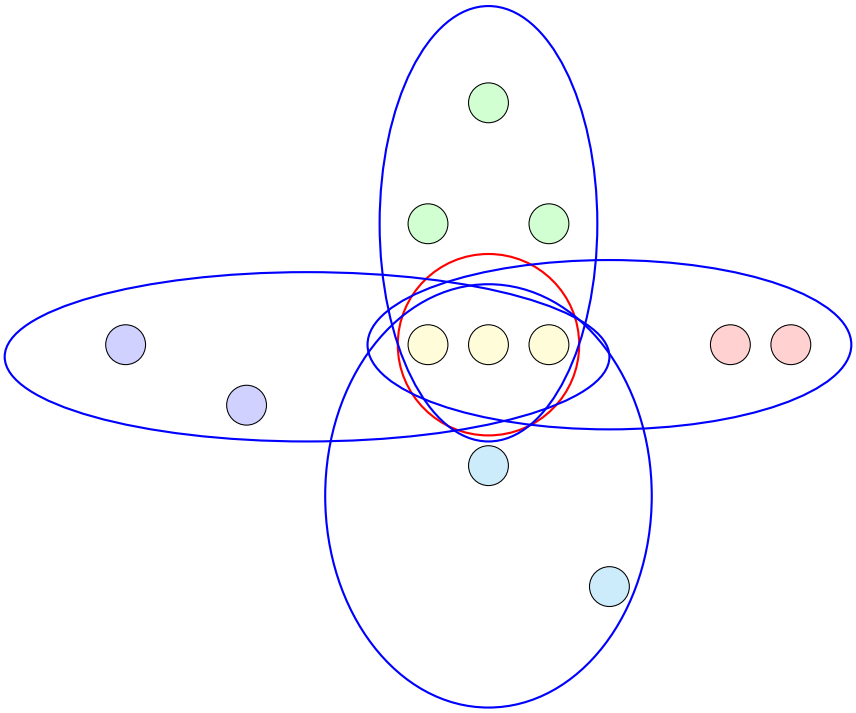


Figure 3.13: Example of a *Sunflower*

Lemma 3.6.1 (Sunflower Lemma)**.** Let $\mathcal{Y}$ be a family of sets (without duplicates) over a universe U, such that all sets have the same cardinality c. If $|\mathcal{Y}| > c!(p-1)^c$, then $\mathcal{Y}$ contains a sunflower with p petals. This sunflower can be computed in polynomial time in $|\mathcal{Y}|, |U|, p$.
Proof. This can be proved by induction on c.

In the d-Hitting Set problem, given a universe U, a family $\mathcal{Y}$ of sets of maximum size d over U and a integer p, the goal is to find if there exists a subset X of U with size p such that X has non empty intersection with every member of $\mathcal{Y}$. The sunflower lemma is used to obtain the kernel of the d-Hitting Set problem.
Now if $\mathcal{Y}$ contains a sunflower $S = \{S_1, S_2, ...S_{p+1}\}$ of cardinality $p+1$, then any hitting set H of $\mathcal{Y}$ of maximum cardinality p, intersects the core X of the sunflower S, as otherwise it would have to intersect $p+1$ petals $S_i \setminus X$.

Reduction 3.6.2. If $(U, \mathcal{Y}, p)$ be an instance of the d-hitting Set. If $\mathcal{Y}$ contains a sunflower S of cardinality $p+1$, then delete $S = \{S_1, S_2, ...S_{p+1}\}$, add a new set $\mathcal{Y}$ along with $U = \bigcup_{X=A'}$, where $A' = A \setminus S \cup X$.

As such using these methods we arrive at a theorem,

Theorem 3.6.3. d-Hitting Set problem admits a kernel with maximum of $d!p^d$ sets and at most $d!p^d \cdot d^2$ elements

Chapter 4

Bounded Search Trees

The concept of bounded search trees is a type of exhaustive search on a problem space. An algorithm which tries to obtain a viable solution through a sequence of choices and for each such choice, considers various possibilities essentially branching a problem into a sequence of subproblems, which are then solved one by one, effectively looks like a search tree, where the required solution is found in one of the leaves. If the size of the search tree is bound by a function of a parameter alone and polynomial time is spent between each node then this branching algorithm runs in FPT time.
Given some measure $\mu(I)$ for instance I, a function of some parameter k alone. In a branch step we generate simpler instances $I_1, I_2, ...I_p (p \geq 2)$ such that following hold true. :

- Every feasible solution S of instance $I_i, i \in \{1, 2, ..., p\}$ corresponds to some feasible solution $h_i(S)$. The set $H(S) = \{h_i(S), i \in \{1, 2, ..., p\}$, S is a feasible solution of $I_i\}$ contains atleast one optimal solution for I.
- p is small and bounded by $\mu(I)$ alone.
- In each branch the problem instance is significantly reduced. That is $\forall i \in \{1, 2, ..., p\}$ we have $\mu(I_i) \leq \mu(I) - c$ where c is some constant.

Original contribution in this chapter is mostly in arrangement and discussion of topics and examples as this chapter is mostly literature review.
We will now see how this concept is used in the Vertex Cover problem.

4.1 Vertex Cover with Bounded Search Trees

Consider a graph G with n vertices and m edges, given an instance (G, k) of the Vertex Cover problem, we have the most simple subset possible to be an edge and as such the problem, becomes trivial to solve when maximum degree of the graph is 1. In case of any larger subsets we have that either a vertex v or all of its neighbours $N(v)$ must be a part of the cover. For an instance (G, k) we have the following recursive algorithm. :

1. if we find vertex v of maximum degree and if v is of degree 1, then any component of G is either an edge or an isolated vertex and thus the solution is trivial.

2. if $|N(v)| \geq 2$ then we have that either v or $N(V)$ belong to the vertex cover.

   - we can delete v and reduce k by 1 to include v in the cover.
   - if $N(V)$ is in the cover then delete we delete the neighbours of V from the graph G and decrease k by $|N(V)|$

In step 2. as we are either reducing our parameter by 1, when v is in the cover or by atleast 2, when $N(v)$ is in the cover, we get our subtrees as $T(i-1), T(i-2)$.

The run time for this algorithm is bounded by (number of nodes in the search tree) $\times$ (time spent at each node). As time taken at each node is bounded by $n^{\mathcal{O}(1)}$. Thus if $T(k)$ is the number of nodes in the search tree then total run time is given by $T(k)n^{\mathcal{O}(1)}$.
Let us define the time bound function $T(k)$ using the following formula :

$$T(i) = \begin{cases} T(i-1) + T(i-2) & \text{if } i \geq 2 \\ 1 & \text{otherwise} \end{cases} \tag{4.1}$$

Now we have that for a recursive relation of the form $T(k) = T(k-d_1) + T(k-d_2) + ... + T(k-d_p)$, where $d_1, d_2, ..., d_p$ is the branching vector, by considering that the bound on number of leaves $T(k)$ to be of the form $T(k) \leq c.\lambda^k$, we can see that recurrence becomes an inequality $\lambda^k \geq \lambda^{k-d_1} + \lambda^{k-d_2} + ... + \lambda^{k-d_p}$ which can be rewritten as a polynomial equation $P(\lambda) = \lambda^d - \lambda^{d-d_1} + \lambda^{d-d_2} ..... - \lambda^{d-d_p} \geq 0$. $P(\lambda)$ is the characteristic polynomial of $T(k)$.

In the above equation 4.1 we get that $T(k) \leq 1.6181^k$, and as such we have that the run time of this algorithm for vertex cover is $\mathcal{O}(1.6181^k n^{\mathcal{O}(1)})$.

We can improve this time drastically by setting a more explicit branching algorithm for vertex cover:

1. If there is some vertex of degree 1, then put its neighbour into the cover and delete it.
2. If there is some vertex v of degree 2, then either both neighbours (w_1, w_2) of v are in the cover or v along with the neighbours of(w_1, w_2) are in the cover.
3. if $|N(v)| \geq 3$ then we recursively branch by considering that either v or $N(V)$ belong to the vertex cover.
   - if v is in the cover then we can delete v and reduce k by 1.
   - if $N(V)$ is in the cover then delete $N(v)$ from G and decrease G by $|N(V)|$

In step 3. as we are either reducing our parameter by 1, when v is in the cover or by atleast 3, when $N(v)$ is in the cover, we get our subtrees as $T(i-1), T(i-3)$. As such we have :

$$T(i) = \begin{cases} T(i-1) + T(i-3) & \text{if } i \geq 3 \\ 1 & \text{otherwise} \end{cases} \tag{4.2}$$

For equation 4.2 we have $T(k) \leq 1.4656^k$, and as such we have that the run time of this algorithm for vertex cover is $\mathcal{O}(1.4656^k n^{\mathcal{O}(1)})$.
As such we see it is possible to get even better run times for vertex cover by making more explicit rules for branching.

4.2 Feedback Vertex Set

A set $F \subseteq V(G)$ for some given graph G is called the feedback vertex set of G if its deletion results in an acyclic graph. The goal of the Feedback Vertex Set problem, is to check if there exists a feedback vertex set of size atmost k in a undirected graph G, where k is a non negative integer.
Here we consider this problem on a more general case of multigraphs for the sake of convenience. Below in decreasing order of priority are reduction rules for Feedback vertex set problem. :

Reduction 4.2.1. If there is a loop at any vertex v, delete that vertex and reduce k by 1.
As loops count as cycles this is required by definition of feedback vertex set.

Reduction 4.2.2. If there is any edge who's multiplicity is larger than 2, then reduce the multiplicity to 2.
As multiplicity of a multiple edge does not affect the set of solutions.

Reduction 4.2.3. Delete any vertex with degree at most 1.
As loops are gone, degree 1 vertices do not contribute to any cycles.

Reduction 4.2.4. Delete any vertex v of degree 2 and connect the neighbours by a new edge.
As loops are gone, we have that the nighbours of v are distinct from v, however it is also possible that v has a double edge, in which case v gets deleted and adds a loop to the double neighbour of v and subsequently by repeated reductions the vertex will be removed.

We observe that the final graph G, after applying reductions 4.2.1-4.2.4 exhaustively has no loops, only single and double edges and has 3 as the minimum degree of a vertex.

Reduction 4.2.5. Terminate the algorithm in the event that $k < 0$ and conclude that (G, k) is a no instance.
This is a stop condition in case our search exceeds the amount needed for our parameter k.

We define $(v_1, v_2, ..., v_n)$ to be an ordering(descending) of $V(G)$ based on their respective degrees i.e $deg(v_1) \geq deg(v_2) \geq .... \geq deg(v_n)$. Now we have that if $V_{3k} = \{v_1, v_2, ..., v_{3k}\}$ and we state the following lemma without a proof.

Lemma 4.2.6. p58 [1] Every Feedback vertex set in G of size at most k contains atleast one vertex V_{3k}.

Using the lemma 4.2.6 we state the algorithm for Feedback Vertex Set problem.
Given some undirected graph G and a non-negative integer k

1. we first apply reduction rules 4.2.1-4.2.5 exhaustively.
2. Now we either obtain a no-instance and the algorithm concludes.
3. If we instead obtain an equivalent instance (G', k') such that G' has minimum degree of at least 3 and $k' \leq k$.
   - we conclude that it is a yes instance if G' is empty.
   - otherwise we move to the next step.
4. Let $V_{3k'}$ be the set of $3k'$ vertices of G' with largest degrees. By lemma **??** we have that every solution W to the feedback vertex set instance (G', k') contains atleast one vertex from $V_{3k'}$. Thus we branch on one of the choice of these vertices and $\forall v \in V_{3k}$ we recursively apply the algorithm to solve FVS on $(G' - v, k' - 1)$.
   - If one of these branches return a solution W' then $W' \cup \{v\}$ is a feedback vertex set of size at most k' for G'
   - Else we have a no instance.
5. At every recursive step we decrease the parameter by 1. thus we prevent the depth of the search tree from exceeding k'. each branch has at most $3k'$ subproblems. Thus the number of nodes is at most $(3k')^{k'} \leq (3k)^k$

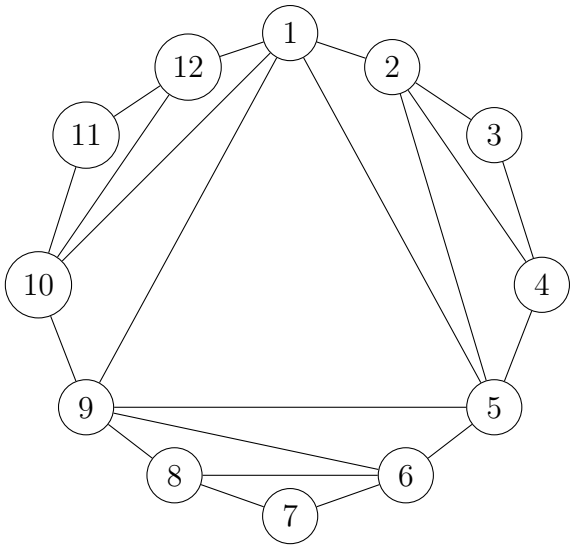
Thus we have

Theorem 4.2.7. For Feedback vertex set there exists an algorithm such that the run time is maximum $(3k)^k \cdot n^{\mathcal{O}(1)}$

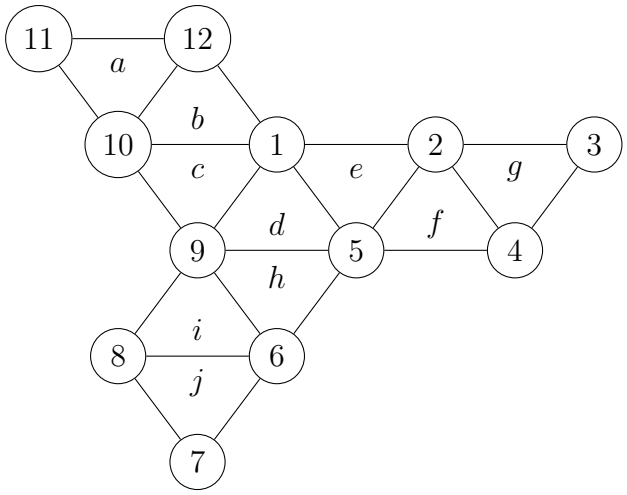
Chapter 5

Treewidth

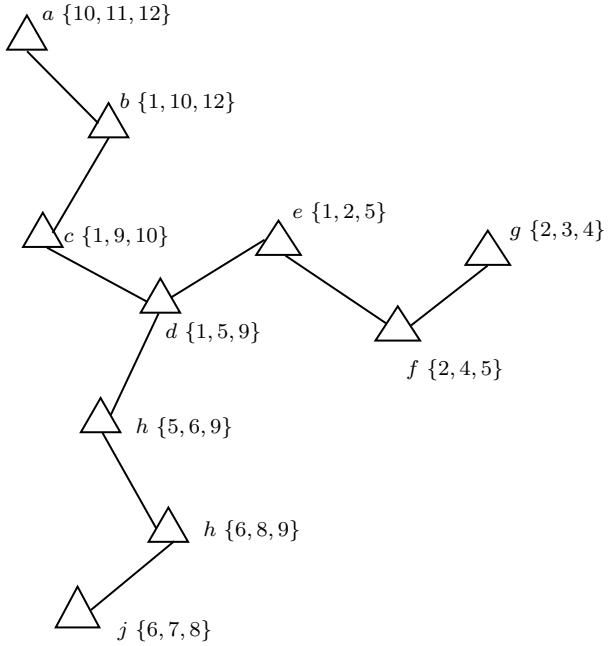
Certain NP-hard problems can be solved when the input has a restricted structure, such as trees, where we can take advantage of special properties of trees. We can implement dynamic programming algorithms on subtrees which represent subproblems that are completely separated. We can then define a class of graphs that generalise trees. The motivation is to find graphs that can be decomposed into disconnected pieces by removing a small number of vertices.



(a) Graph G.



(b) Graph G depicted as a composition of 10 interlocking triangles.



(c) Graph G decomposed into a tree-like structure, with each triangle corresponding to a node.

Figure 5.1: Parts (a) and (b) depict alternate drawings of graph G drawn, while part (c) shows how the triangles form a tree-like structure.

The graph G shown in Figure 5.1a can be decomposed to a tree-like structure, as shown. This is not very clear from (a). In (b), G is viewed as being composed of 10 interlocking triangles. Deletion of triangles a, g, j that are at the extremities is similar to deletion of the

leaves of a tree, while deleting any of the seven remaining triangles results in G falling apart into disconnected pieces. In (c), By having each triangle correspond to a node in a tree, we observe that their structure is tree-like. Notice that the same vertices of the graph G occur in multiple triangle, which are not necessarily adjacent; and there are edges between vertices in triangles that are distant in the tree-structure.

5.1 Tree Decomposition

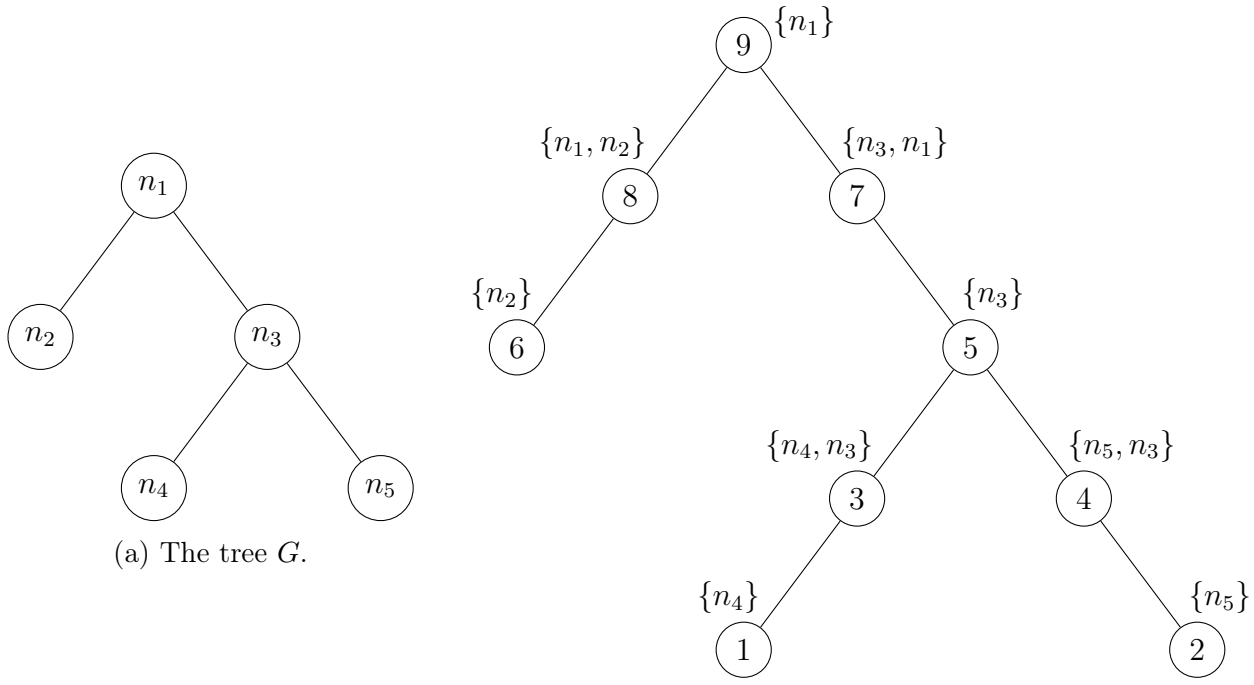
Definition 5.1.1. A **tree decomposition** of a graph $G = (V, E)$ is an ordered pair $(T, \{V_t : t \in T\})$ that consists of a tree T (with node set varying from that of G), and a subset $V_t \subseteq V$ for every node t of T. The subsets V_t are called the "bags" or "pieces" of the tree decomposition. The tree T and the collection of bags $\{V_t : t \in T\}$ must satisfy the following properties :

1. **Node Coverage.** Every vertex of G must belong to at least one bag V_t.

2. **Edge Coverage.** Given any edge e of G, there is some bag V_t that contains both ends of e.

3. **Coherence.** Given three nodes t_1, t_2 and t_3 of T satisfying the condition that t_2 lies on the path from t_1 to t_3. Then, if a vertex v of G belongs to both V_{t_1} and V_{t_3}, then it also belongs to V_{t_2}.

When the graph G is a tree, the tree decomposition is constructed as follows : For every vertex v and every edge e of G, the decomposition tree T has a node t_v and a node t_e, respectively. The tree T has and edge (t_v, t_e) when v is an endpoint of e. For every vertex v in G, we define the bag $V_{t_v} = \{v\}$, and for every edge $e = (u, v)$ in G, we define the bag $V_{t_e} = \{u, v\}$. Below is an example of a tree decomposition for a tree G.



(a) The tree G.

(b) Tree decomposition T of G.

Figure 5.2: The tree decomposition T of a tree G as described above.

Definition 5.1.2. A tree decomposition $(T, \{V_t\})$ of G such that there is no edge (x, y) of T such that $V_x \subseteq V_y$ is called a **non-redundant** tree decomposition of a graph G

For some tree decomposition $(T, \{V_t\})$ of a graph G, we make it non-redundant as follows : If T has an edge (x, y) , such that $V_x \subseteq V_y$, then we contract the edge (x, y) by folding the bag V_x into the bag V_y and obtain a tree decomposition of G based on a smaller tree.

5.2 Properties of a Tree Decomposition

Trees have two nice separation properties :

1. deleting an edge e from a tree causes it to split into exactly two connected components.

2. deleting a node t from a tree causes it to split into number of components equal to the degree of t.

The Coherence Property of a tree decomposition leads to tree-like separation properties.

Notation. Let T' be a subgraph of T. $G_{T'}$ denotes the subgraph of G induced by all the nodes in all the bags associated with the nodes of T', i.e., the set $\bigcup_{t \in T'} V_t$.

Theorem 5.2.1. Let $T_1, \ldots, T_d$ be the components of T after deletion of a node t with bag V_t. Then the subgraphs $G_{T_1} - V_t, G_{T_2} - V_t, \ldots, G_{T_d} - V_t$ of G share no common vertices, and do not have edges between them.

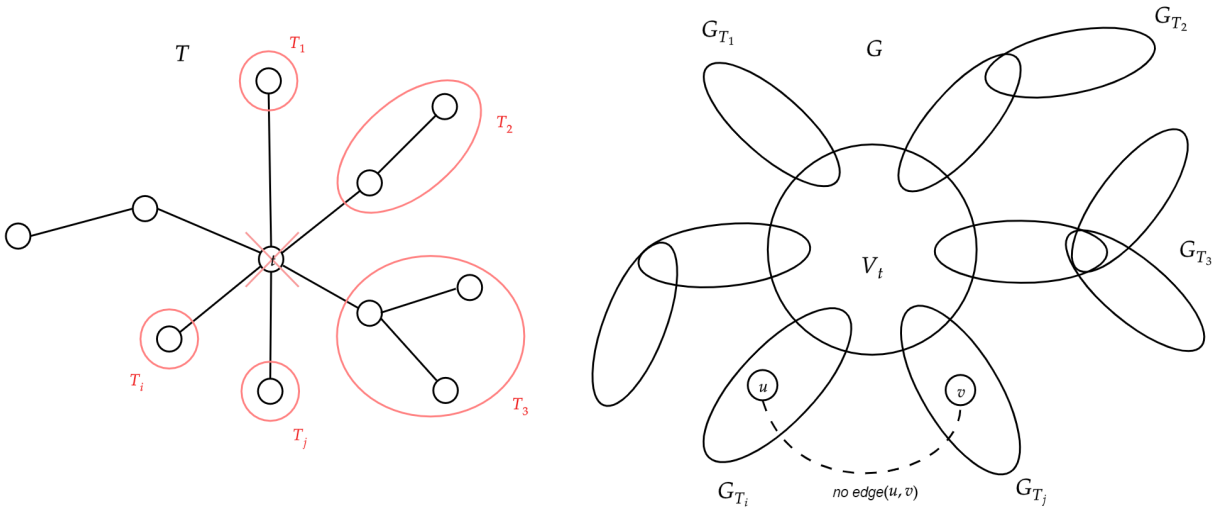


Figure 5.3: Separations of the tree T translate to separations of the graph G

Proof. Let us assume that the subgraphs $G_{T_i} - V_t$ and $G_{T_j} - V_t$ $(i \neq j)$ share a vertex v. Then,

$$v \in V_x \text{ for some } x \in T_i, \text{ and } v \in V_y \text{ for some } y \in T_j.$$

Since t lies on the $x - y$ path in T, it follows from the Coherence Property that $v \in V_t$. But then this contradicts our assumption that $v \in G_{T_i} - V_t$, and $v \in G_{T_j} - V_t$. Thus, the subgraphs $G_{T_i} - V_t$ do not share any vertices.

Now let us assume there is some edge $e = (u, v)$ in G, with one end u in the subgraph $G_{T_i} - V_t$ and the other end v in the subgraph $G_{T_j} - V_t$ for some $i \neq j$. By the Edge Coverage Property, $\exists V_x$ such that $u, v \in V_x$. Both the subgraphs T_i and T_j cannot have the node x. Without loss of generality, suppose $x \notin T_i$. The vertex u is in the subgraph $G_{T_i} - V_t$,

$\Rightarrow u \in V_y$ for some $y \in T_i$

$\Rightarrow u \in V_x$ and V_y .

Since the $x - y$ path in T has t, it follows from the Coherence Property that $u \in V_t$. This contradicts our assumption that $u \in G_{T_i} - V_t$. Thus there is no edge $e = (u, v)$ in G with one end u in the subgraph $G_{T_i} - V_t$ and the other end v in the subgraph $G_{T_j} - V_t$ for some $i \neq j$. □

Example 7. Figure 5.4 gives an example of a tree decomposition of width 2. Let $V_t = \{c, d, h\}$. If we remove this bag from the graph we will have three components. First component is the subgraph induced by a, b. Secoond component is the subgraph induced by g, f. The third component is the subgraph induced by e only.

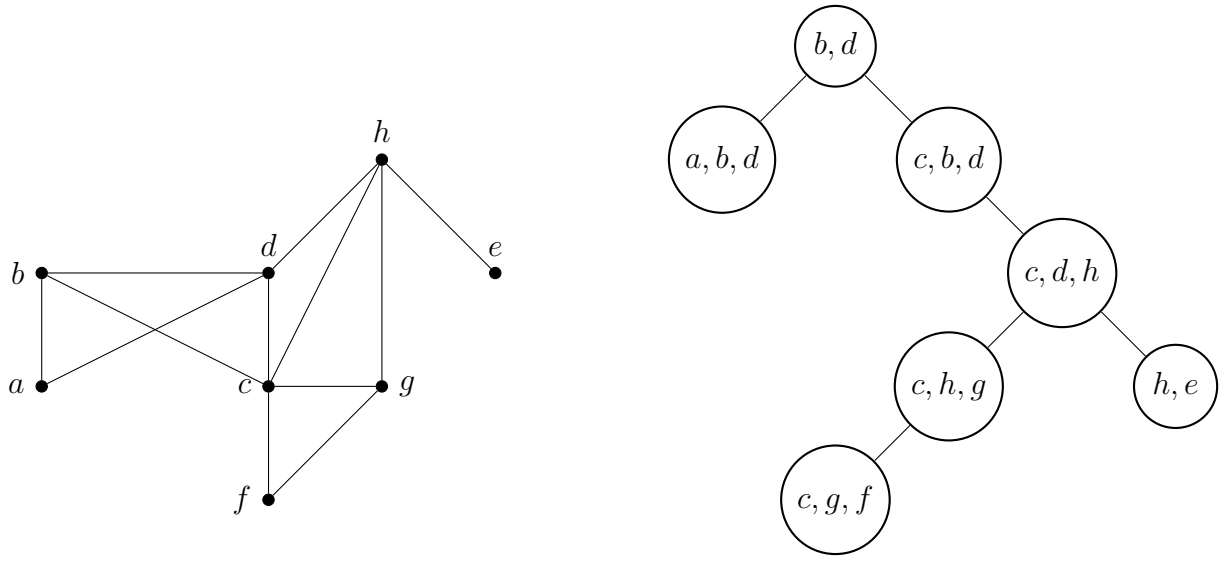


Figure 5.4: Example of a tree decomposition of width 2

Theorem 5.2.2. *Let T be a tree decomposition of graph G. Suppose A and B are two components of tree decomposition T once we delete edge (a, b) from T. Then deletion of the vertices of intersection of two bags V_a and V_b, that is, removing the vertices of the set $V_a \cap V_b$ disconnects G into the two subgraphs $G_A - (V_a \cap V_b)$ and $G_B - (V_a \cap V_b)$. In particular, these two subgraphs do not have any common vertices, and there is no edge with one end in the first subgraph $G_A - (V_a \cap V_b)$ and the other end in the second subgraph $G_B - (V_a \cap V_b)$.*

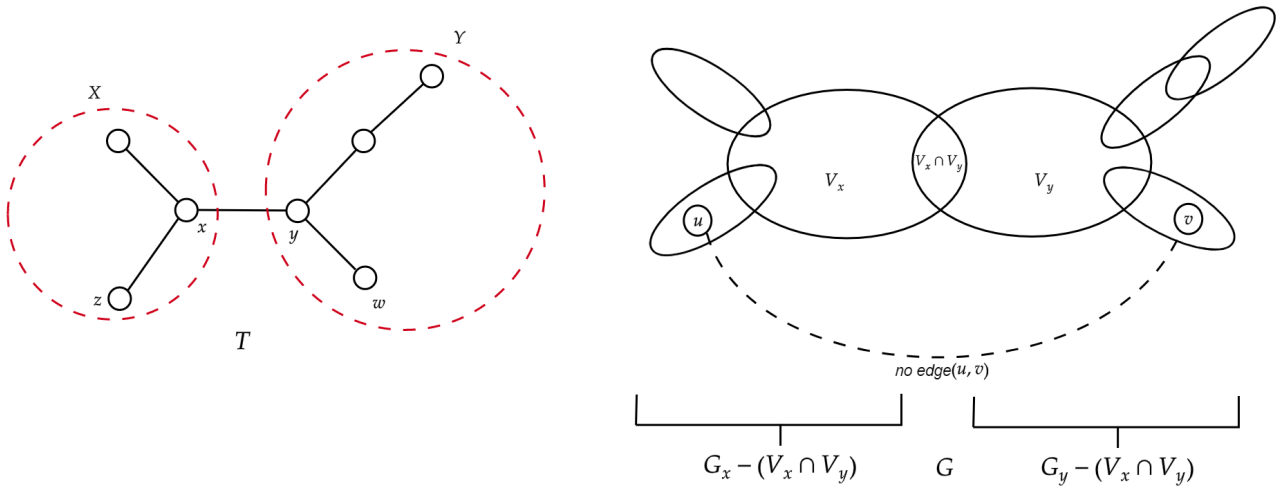


Figure 5.5: Deleting an edge of the tree T translates to separation of the graph G

Proof. Suppose, for the sake of contradiction, that these two subgraphs $G_A - (V_a \cap V_b)$ and $G_B - (V_a \cap V_b)$ share a vertex v. Then,

$$v \in V_\alpha \text{ for some } \alpha \in A, \text{ and } v \in V_\beta \text{ for some } \beta \in B.$$

Since both a and b lie on the $\alpha - \beta$ path in T, it follows from the Coherence Property that $v \in V_a$, and $v \in V_b$.

$\Rightarrow v \in V_a \cap V_b$.

This contradicts our assumption that $v \in G_A - (V_a \cap V_b)$, and $v \in G_B - (V_a \cap V_b)$.
Hence, the two subgraphs $G_A - (V_a \cap V_b)$ and $G_B - (V_a \cap V_b)$ in graph G do not have any

common vertices.

Now let us assume that there is an edge $e = (u, v)$ in G, having an end u in the subgraph $G_B - (V_a \cap V_b)$, and the other end v in the subgraph $G_A - (V_a \cap V_b)$. By the Edge Coverage Property, there exists a bag V_z such that $u, v \in V_z$. Suppose by symmetry that $z \in A$. The vertex $v \in G_B - (V_a \cap V_b)$,

$\Rightarrow v \in V_w$ for some $w \in B$.

Since both a and b lie on the $z - w$ path in T, it follows from the Coherence Property that $v \in V_a$, and $v \in V_b$.

$\Rightarrow v \in V_a \cap V_b$.

This contradicts our assumption that $v \in G_B - (V_a \cap V_b)$.
Hence, there does not exist an edge $e = (u, v)$ in G, having an end u in $G_B - (V_a \cap V_b)$, and the other end v in $G_A - (V_a \cap V_b)$. □

Example 8. Figure 5.4 shows a tree decomposition T of the graph G shown at the left. Consider two adjacent bags $V_a = \{c, b, d\}$ and $V_b = \{c, d, h\}$ in the tree decomposition T. If we remove the edge joining the bags $V_a = \{c, b, d\}$ and $V_b = \{c, d, h\}$, then we will have two components: the first component contains bags $\{a, b, d\}, \{b, d\}, \{c, b, d\}$ and the second component contains bags $\{c, d, h\}, \{c, h, g\}, \{c, g, f\}, \{he\}$. Therefore $A = \{a, b, c, d\}$ and $B = \{c, d, h, g, f, e\}$ and G_A is the subgraph of G induced by A and G_B is the subgraph of G induced by B. Here $G_A - (V_a \cap V_b) = \{a, b\}$ and $G_B - (V_a \cap V_b) = \{f, g, h, e\}$. It is easy to observe that there is no edge between subgraphs induced by $G_A - (V_a \cap V_b) = \{a, b\}$ and $G_B - (V_a \cap V_b) = \{f, g, h, e\}$.

Every graph has a trivial tree decomposition where the tree decomposition T consists of a single node t and the bag V_t contains the entire vertex set of the graph. We look for tree decompositions where the bags are *small*. Since tree decompositions have tree-like separation properties, having smaller bags implies that the deletion of a very small set of vertices of the original graph breaks the graph apart into disconnected subgraphs (components).

Definition 5.2.1. Given a tree decomposition $(T, \{V_t\})$ of some graph G we define it's **width** to be:

$$width(T, \{V_t\}) = \max_t |V_t| - 1$$

The minimum width among all possible tree decompositions of G is defined to be the **tree-width** of G.

All tree decompositions must have at least one bag with two vertices due to Edge Coverage Property, and hence a graph must have tree-width ≥ 1.

Remark 5.2.1. *Let H be a subgraph of G. Then the tree-width of H cannot exceed the tree-width of G.*

We can get a tree decomposition of H with $tw(H) \leq tw(G)$, starting from a tree decomposition of G the following way. Suppose $(T, \{V_t\})$ is a tree decomposition of G. We can construct a tree decomposition of H by preserving the tree T and replacing the bags V_t with $V_t \cap H$. For example, consider the tree decomposition in Figure 5.4. Let $H = G - \{c\}$. Then the bags of the tree decomposition of H are $\{a, b, d\}, \{b, d\}, \{d, h\}, \{h, e\}, \{h, g\}, \{g, f\}$. Clearly, the treewidth of H is less than or equal to the treewidth of G.

Theorem 5.2.3. *Let G be a connected graph. Then G has tree-width one if and only if it is a tree.*

Proof. ($\Rightarrow$) Under Definition 1, we have stated a method to construct a tree decomposition of tree-width 1 for a tree G.

($\Leftarrow$) Now suppose, if possible, that some graph G which is not a tree is a connected graph of tree-width 1. Since G is not a tree, it has a subgraph consisting of a simple cycle C. By using Remark 5.2.1, it suffices to show that the graph C does not have tree-width 1.
Suppose, if possible, that C had a tree decomposition $(T, \{V_t\})$ in which each bag had size at most 2, i.e,

$$|V_t| \leq 2 \quad \forall t \in T.$$

Without loss of generality, we can assume that T is non-redundant. Let (x, y) be any edge in T. Delete (x, y). Define X and Y to be the components of $T - (x, y)$ containing x and y, respectively. Because of non-redundancy, we know that neither V_x nor V_y can equal $V_x \cap V_y$.

$\Rightarrow |V_x \cap V_y| \leq 1$.

Deleting $V_x \cap V_y$ separates C into $C_Y - (V_x \cap V_y)$ and $C_X - (V_x \cap V_y)$. Thus both of these subgraphs must be non-empty (as they contain x and y, respectively). But it is not possible to disconnect a cycle into two non-empty subgraphs by deleting a single vertex (as $|V_x \cap V_y| = 1$). This yields a contradiction.

Theorem 5.2.4. *Any non-redundant tree decomposition of an n-vertex graph has a maximum of n bags.*

Proof. By induction on n.
The case $n = 1$ is obvious. Consider the case $n > 1$:
Given a non-redundant tree decomposition $(T, \{V_t\})$ of an n-vertex graph G, choose a leaf t of T. By non-redundancy, there exists at least one vertex in V_t that does not occur in the neighbouring bag, and thus by the Coherence Property, this vertex does not occur in any other bag. Let U be the set of all such vertices in V_t.

On deleting t from T and removing V_t from the collection of bags, we obtain a non-redundant tree decomposition of $G - U$. By the induction hypothesis, this new tree decomposition has $n - |U| \leq n - 1$ bags. Therefore, the original tree decomposition $(T, \{V_t\})$ has at most n bags.

Theorem 5.2.5. *An n-vertex graph of tree-width at most k has at most kn edges.*

Proof. By induction on n.
The case $n = 1$ is obvious. Consider the case $n > 1$:
Given a non-redundant tree decomposition $(T, \{V_t\})$ of an n-vertex graph G, choose a leaf t of T. Let U be the set of all such vertices in V_t that do not occur in the neighbouring bag, and thus by the Coherence Property, they do not occur in any other bag. By non-redundancy, $|U| \geq 1$.

On deleting t from T and removing V_t from the collection of bags, we obtain a non-redundant tree decomposition of $G - U$. By the induction hypothesis, the number of edges in $G - U$ is at most $k(n - |U|)$.
By the Edge Coverage Property, each vertex $v \in U$ can only have neighbours in V_t, and hence its degree is at most $|V_t| - 1 \leq k$. Hence, G has at at most $k|U|$ edges more than $G - U$. Thus G has at most kn edges.

5.3 Dynamic Programming over a Tree Decomposition

In this section we present two examples to illustrate how dynamic programming is useful to design efficient algorithms on graphs of bounded treewidth. We consider the maximum weighted independent set problem and dominating set problem. The input to such problems is an undirected graph and its tree decomposition.

5.3.1 Maximum-Weight Independent Set

Given a graph $G = (V, E)$, and a weight $w_v > 0$ associated with each vertex $v \in V$. The Maximum-Weight Independent Set Problem is to find an independent set S in the graph G so that the total weight $\sum_{v \in S} w_v$ is maximum.

If the graph given is a tree T, we designate the root T and construct the independent set by working our way up from the leaves. For a node u, let T_u denote the subtree formed by u and all its descendants, with u as the root. We aim to solve the subproblem associated with the subtree T_u after having solved the subproblems for all its children. For every internal node u, to get a maximum-weight independent set S for the tree T_u, we make a decision whether to include u in S or not. Once this decision is made, the problems for the different subtrees below u become independent. This leads to a recurrence relation which yields a dynamic programming algorithm by building up the optimal solutions over larger and larger subtrees.

This process can easily be generalized for a graph G having a tree decomposition $(T, \{V_t\})$ with width w. We designate the root of the tree T and construct the independent set by starting at the leaves and moving upwards by considering the bags V_t. At an internal node t

of T, the optimal independent set S and the bag V_t intersect in some subset U. We enumerate all possible ways to obtain such a subset from the vertices in V_t. Since $|V_t| \leq w+1$, there are a total of 2^{w+1} possibilities to consider. There are two advantages of this :

1. 2^{w+1} possibilities for every bag is a lot more reasonable than 2^n total possibilities of S, when w is much smaller than n.
2. Once we have determined the set $U \subseteq V_t$, the separation properties proven in Theorems 1 and 2 ascertain that we can independently solve the problems in the different subtrees of T below t.

Given that individual bags are reasonably small, although the algorithm employs a brute-force search at a level of a single bag it is quite efficient at the global scale.

Subproblem Definition

- We designate the root of the tree T at some node r. For any node $t \in T$, T_t denotes the subtree with the root as t. By the previous notation, G_{T_t} represents the subgraph of G induced by the vertices in all the bags related to nodes of T_t. For notational simplicity, we denote G_{T_t} by G_t.
  For any $U \subseteq V$, $w(U) = \sum_{u \in U} w_u$ denotes the sum of the weights of the vertices in U.
- For every subtree T_t, we create a set of subproblems, such that each corresponds to a possible set $U \subseteq V_t$, such that the intersection of the optimal solution with V_t is represented. This implies that U has to be an independent set as it cannot represent the intersection of V_t with the optimal solution.
  Thus, for any independent set $U \subseteq V_t$, we define the maximum weight of an independent set S in G_t to be $f_t(U)$, provided $S \cap V_t = U$.
- Since for a node t of T, we have that number of independent subsets of V_t is at most 2^{w+1}, there are a maximum of 2^{w+1} subproblems associated with each node. By Theorem 4, we can assume that the tree decomposition has at most n bags. Hence, the total number of subproblems to solve is $2^{w+1}n$.
  Once all these subproblems have their solutions, for an independent set in G, the maximum weight can be determined by considering the subproblems related to the root r. The value is

$$\max\{f_r(U) : U \subseteq V_r \text{ is independent}\}.$$

Building up Solutions We desire a recurrence relation to build up the solutions for the subproblems discussed above.

If t is a leaf, then we have that $f_t(U) = w(U)$ for all independent sets $U \subseteq V_t$.

Suppose $t_1, \ldots, t_d$ are the children of t, and the values of $f_{t_i}(W)$, $1 \leq i \leq d$ for each independent set $W \subseteq V_{t_i}$ have already been determined. For an independent set $U \subseteq V_t$, we need to calculate the value of $f_t(U)$.

Let S be the maximum-weight independent set in G_t subject to the constraint $S \cap V_t = U$, that is,

$$w(S) = f_t(U).$$

It is important to understand how the set S looks when it intersects with each of the subgraphs G_{t_i}. Let the intersection of S with the vertices of G_{t_i}, be denoted by S_i. Refer to Figure 5 below.

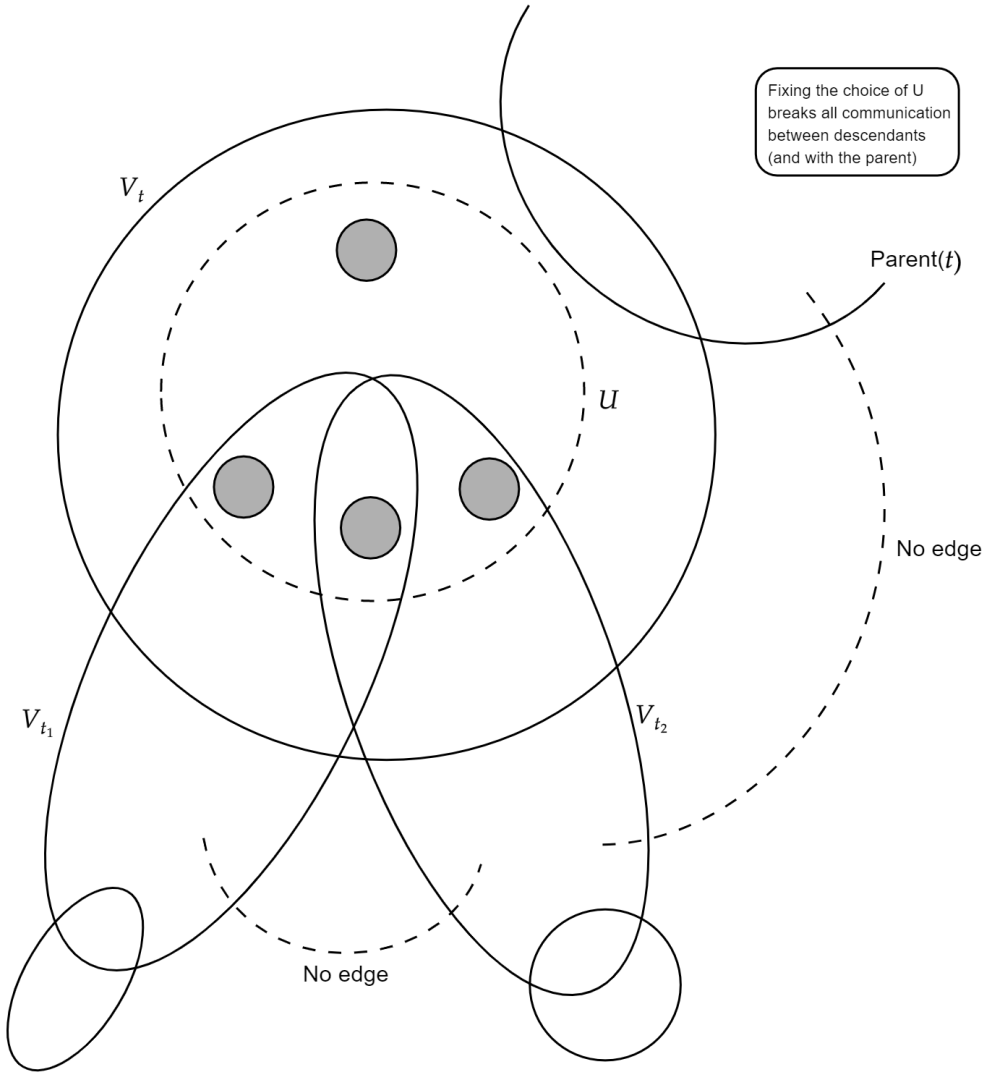


Figure 5.6: The subproblem $f_t(U)$ in the subgraph G_t. We consider independent sets S_i in the descendent graphs G_{t_i}, subject to the constraint $S_i \cap V_t = U \cap V_{t_i}$, for the optimal solution to this subproblem.

Let W_{t_i} denote the union of the bags below V_{t_i}, including V_{t_i}.

$\Rightarrow S_i = S \cap W_{t_i}$

Thus $S_i \cap V_t = U \cap W_{t_i}$, and $W_{t_i} \cap U = V_{t_i} \cap U$, because a vertex in that is in both W_{t_i} and V_t has to be in V_{t_i} too, by the Coherence Property. Thus,

$$S_i \cap V_t = U \cap V_{t_i}.$$

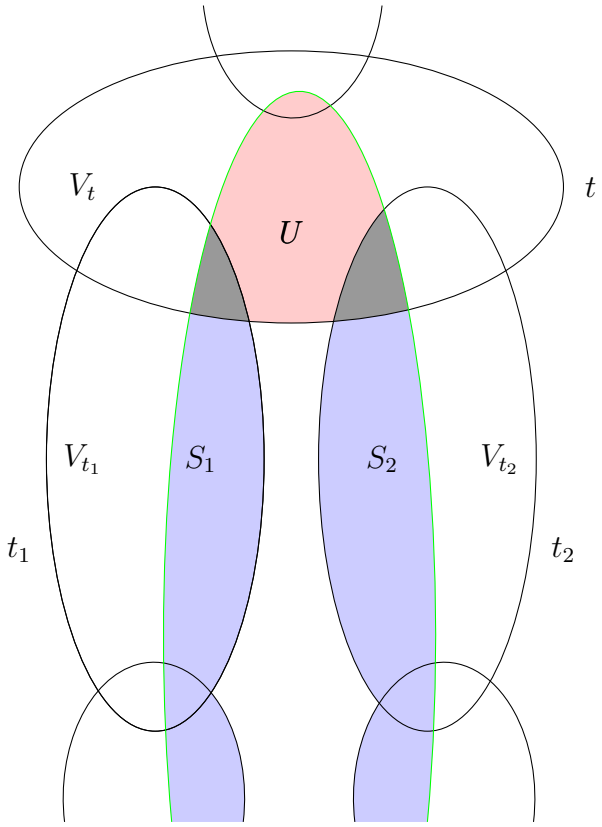


Figure 5.7: The green line delimits the set S. The red shaded area represents the set $U \subseteq V_t$. The blue shaded area represents S_i for $i = 1, 2$. The grey shaded area represents $S_i \cap V_t = U \cap V_{t_i}$, for $i = 1, 2$.

Theorem 5.3.1. *S_i is a maximum-weight independent set of G_{t_i}, subject to the constraint $S_i \cap V_t = U \cap V_{t_i}$.*

Proof. Assume there is an independent set S_i' of G_{t_i}, satisfying the relation $S_i' \cap V_t = U \cap V_{t_i}$ and $w(S_i') > w(S_i)$. Consider the set $S' = (S - S_i) \cup S_i'$. Clearly, $w(S') > w(S)$. Note that,

$$\begin{aligned} S' \cap V_t &= ((S - S_i) \cap V_t) \cup (S_i' \cap V_t) \\ &= (S \cap V_t) - (S_i \cap V_t) \cup (S_i' \cap V_t) \\ &= (S \cap V_t) - (U \cap V_{t_i}) \cup (U \cap V_{t_i}) \\ &= S \cap V_t \\ &= U \end{aligned}$$

Now we claim that S' is an independent set in G_t. For the sake of contradiction, assume

that S' is not an independent se. Let $e = (u, v)$ be an edge such that both ends are in S'. It is not possible that v and u both belong to S'_i, or that they both belong to S. This is because both S and S'_i are independent sets.

Thus, without loss of generality, we can say $u \in S - S'_i$ and $v \in S'_i - S$. This implies that u is not a vertex in the subgraph G_{t_i}, while v is a vertex in the subgraph $G_{t_i} - (V_t \cap V_{t_i})$. By Theorem 5.2.2, there cannot be an edge in S' joining u and v. Figure 5.7 illustrates the above argument.

The above claim tells us that S' is an independent set in G_t. But this contradicts the choice of the maximum-weight independent set in G_t as S. Thus, the set S'_i as described in the beginning of the proof cannot exist. This proves that S_i is a maximum-weight independent set of G_{t_i}, subject to the constraint $U \cap V_{t_i} = S_i \cap V_t$. □

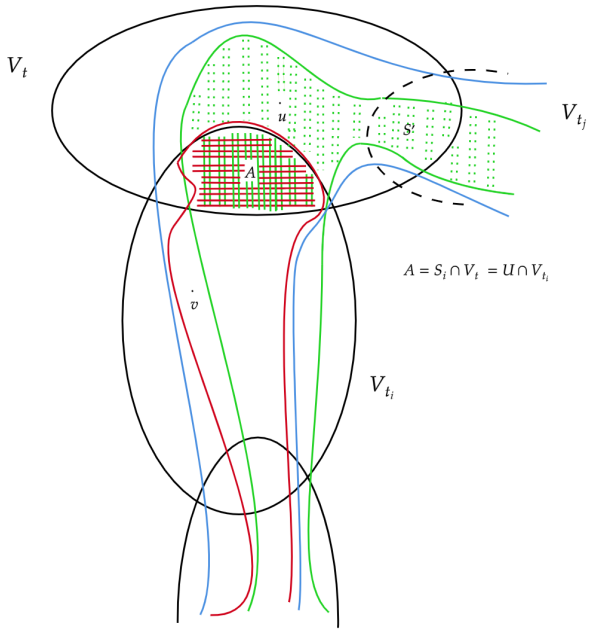


Figure 5.8: The green line delimits the set S. The green shaded region represents the set $S' = (S - S_i)$. The red line delimits the set S_i. The blue line delimits the set $S' = (S - S_i) \cup S'_i$. The locations of the vertices u and v are shown.

Theorem 5.3.1 is essential for designing a recurrence relation for the subproblems. It tells us that the information needed to compute $f_t(U)$ is implicit in the values already computed for the subtrees below t.

For each child t_i $(1 \leq i \leq d)$, we just need to determine the value of the maximum-weight independent set S_i in G_{t_i} subject to the constraint $S_i \cap V_t = U \cap V_{t_i}$. We build the solution by solving smaller subproblems followed by applying the recurrence relation. Thus, while looking at the child t_i, we only need to consider the set $S_i \cap V_{t_i}$. But the constraint states that it can be any independent set $U_i \subseteq V_{t_i}$ such that $U \cap V_{t_i} = U_i \cap V_t$ instead of entirely determining what $S_i \cap V_{t_i}$ should be.

Therefore, the optimal set S_i in G_{t_i} has weight equal to

$$\max\{f_{t_i}(U_i) : U_i \cap V_t = U \cap V_{t_i} \text{ and } U_i \subseteq V_{t_i} \text{ is independent}\}.$$

The value of $f_t(U)$ these maxima added over the d children of t plus is $w(U)$. However, we exclude the vertices that are common to both the sets U and U_i from the contribution of the children to avoid overcounting them in U. Thus,

$$f_t(U) = w(U) + \sum_{i=1}^{d} \max\{f_{t_i}(U_i) - w(U \cap U_i) : U_i \cap V_t = U \cap V_{t_i} \text{ and } U_i \subseteq V_{t_i} \text{ is independent}\}. \tag{5.1}$$

The final algorithm builds up the values of all the subproblems starting upwards from the leaves of T. By tracing back through the execution of the algorithm, one can find an actual independent set of maximum weight.

Time Complexity

- For each of the d children t_i of a node t of T, and for each independent set $U_i \subseteq V_{t_i}$, $O(w)$ time is needed to check if $U \cap V_{t_i = U_i \cap V_t}$, to determine if it should be considered in the computation of (1).

- There is a maximum of 2^{w+1} sets U_i associated with a child t_i, $1 \leq i \leq d$. Thus the total time required for computing $f_t(U)$ is $O(2^{w+1}wd)$.
- There is a maximum of 2^{w+1} sets U related to a node t, so $O(4^{w+1}wd)$ gives us the total time spent on node t.
- This time is summed over all nodes t of T to get the total running time of the algorithm for an n-vertex graph G with tree decomposition $(T, \{V_t\})$. The sum of the number of children of a node t over all nodes t of T is $O(n)$ as each node is counted as a child exactly once. Thus the total running time is $O(4^{w+1}wn)$.

Chapter 6

Conclusions

In this survey of techniques we have covered three core topics of fixed-parameter algorithms, namely Kernelization, Bounded search trees and Treewidth and have studied the concept of parameterized Vertex Cover problem thoroughly.

We observe that with correct parameterization of a problem, the problem becomes simpler to solve. That is the correct choice of the kernel or parameter is important and as such decides how simple the FPT becomes.

We have shown that FPT is an essential concept in algorithms and algorithm designs.

With sufficient understanding of these topics it is possible to study W-hierarchy and attempt understanding problems such as minimum k-union problem, small set edge expansion problem, small set vertex expansion problem, small set bipartite vertex expansion problem, etc.

Bibliography

[1] Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk and Saket Saurabh, *Parameterized Algorithms*. Springer, 2016. <http://parameterized-algorithms.mimuw.edu.pl/>

[2] Fedor V. Fomin and Saket Saurabh, "Kernelization Methods for Fixed-Parameter Tractability." *Tractability: Practical Approaches to Hard Problems*, Cambridge, 2014. 260-282.

[3] Rodney G. Downey and Michael R. Fellows *Fundamentals of Parameterized Complexity*, Springer, 2013.

[4] R. Diestel, *Graph Theory* 5th edition. Springer, 2017.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to algorithms*, The MIT Press, 2001.

[6] Falk Hüffner, Rolf Niedermeier, and Sebastian Wernicke. (2008). "Techniques for Practical Fixed-Parameter Algorithms" *The Computer Journal* 51. DOI: 10.1093/comjnl/bxm040.

[7] Éva Tardos and Jon Kleinberg. *Algorithm Design*, Pearson, 2005.