The Minimum Neighbourhood Problem

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

Chinmay Joshi

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

April, 2019

Supervisor: Dr. Soumen Maity

Certificate

This is to certify that this dissertation entitled The Minimum Neighbourhood Problem 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 Chinmay Joshi at Indian Institute of Science Education and Research under the supervision of Dr. Soumen Maity, Associate Professor, Department of Mathematics, and Dr. Saket Saurabh, Professor, TCS group, Institute of Mathematical Sciences, during the academic year 2018-2019.

S Saurabh 1/.　　　Soumen Maity

Dr. Saket Saurabh　　　Dr. Soumen Maity

Committee:

Dr. Soumen Maity

Dr. Saket Saurabh

Dr. Geevarghese Philip

This thesis is dedicated to my parents

Declaration

I hereby declare that the matter embodied in the report entitled The Minimum Neighbourhood Problem 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.

Chinmay Joshi.

Chinmay Joshi

Acknowledgments

I would first like to thank Dr. Soumen Maity. He consistently allowed this to be my own work, but steered me in the right direction whenever he thought I needed it. I would also like to thank Dr. Saket Saurabh from the Institute of Mathematical Sciences. The door to his office was always open whenever I had a question about my work. I would also like to acknowledge Dr. Geevarghese Philip from The Chennai Mathematical Institute as the second reader of this thesis, and I am gratefully indebted to him for his very valuable comments on this thesis. Finally, I must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Abstract

Given a graph $G = (V, E)$ with n vertices and a positive integer $s \leq n$, we want to find a set $S \subseteq V$ of size s such that $|N_G[S]|$ is minimum, where $N_G[S]$ denotes closed neighbourhood of S. We call this problem as the minimum neighbourhood problem (MNP). In this project, we give a parameterized algorithm which takes as input a graph G, its tree decomposition with width at most k, and a positive integer s, and returns $|N[S]|$ such that $S \subseteq V$, $|S| = s$ and S has minimum neighbours in G, where the parameter is k.

Contents

Abstract **xi**

Introduction **1**

1 Preliminaries **5**

1.1 Weighted Independent Set 9

2 NP-completeness of the minimum neighbourhood problem **11**

3 Minimum Neighbourhood Problem **13**

4 Conclusions **19**

Appendices **21**

A Python Code **23**

Introduction

The neighbourhood of v, written $N_G(v)$, is the set of vertices adjacent to v in G; and $N[v] = N(u) \cup \{v\}$ denotes the closed neighbourhood of v. For a subset $S \subseteq V(G)$, we use $N_G[S] = \cup_{v \in S} N_G[v]$, to denote the closed neighbourhood of S in G. The input to the parameterized version of Minimum Neighbourhood Problem is a graph G with two integers $s, \ell \leq |V(G)|$, and (G, s, ℓ) is a yes-instance if G has a set S of s vertices such that $|N_G[S]| \leq \ell$.

Computational problems are classified on the basis of their complexity. To decide how complex a problem is, a generally accepted standard is the time in which it can be solved by an algorithm. An "efficient" algorithm is one that runs in time polynomial in the size of input, to yield the solution. The problems that are solvable by polynomial time algorithms are considered "easy" and those that require super polynomial time algorithms are deemed "hard".

The computationally hard problems are classified as NP-hard. These problems are neither known to have a polynomial time solution, nor has anyone been able to prove that such a solution does not exist. Several problems that do occur in practice, are NP-hard. The best known algorithms that are used to solve them, require exponential time or worse. Some approaches to tackle these problems are approximation and parameterization. Approximation algorithms are those that run in polynomial time to yield a solution that is closed to the optimum. In this technique, we relax the constraint of optimality and can therefore aim for a polynomial time solution.

A relatively recent approach to solving NP-hard problems is parameterization [3, 4]. A parameterized problem has an input instance x, as well as a parameter k, which is believed to be sufficiently small compare to the size of input instance. The art of parameterization lies in selecting the best possible parameter such that our algorithm is efficient, and the computational explosion is restricted to the parameter. Some NP-hard and NP-complete problems can be solved by algorithms that are exponential in the size of a fixed parameter while polynomial in the size of the input. Such problems are called fixed parameters tractable (FPT). FPT contains the fixed parameter tractable problems, which are those that can be solved in time $f(k)|x|^{O(1)}$ for some computable function f.

Definition 0.0.1. *[3] A parameterized problem* is a language $L \subseteq \Sigma^* \times N$, where Σ is a fixed, finite alphabet. For instance $(x, k) \in \Sigma^* \times N$, k is called the parameter.

For example, in parametrized algorithm, the problem of finding minimum vertex cover in graph G translates to whether there exists a vertex cover of size at most k in G, where k is the parameter.

Definition 0.0.2. *[3]*A parameterized problem $L \subseteq \Sigma^* \times N$ is called *fixed-parameter tractable* (FPT) if there exists an algorithm A (called a fixed-parameter algorithm), a computable function $f : N \to N$, and a constant c such that, given $(x, k) \in \Sigma^* \times N$ the algorithm A correctly decides whether $(x, k) \in L$ in time bounded by $f(k)|(x, k)|^c$. The complexity class containing all fixed-parameter tractable problems is called FPT.

For example, the vertex cover problem is FPT. By using kernalization algorithms and reduction methods, the vertex cover problem can be solved in $O(n\sqrt{m} + 1.4656^k k^{O(1)})$ where n and m are the number of vertices and edges in G respectively and k is the parameter.

Definition 0.0.3. *[3]* (XP) A parameterized problem $L \subseteq \Sigma^* \times N$ is called *slice-wise polynomial (XP)* if there exists an algorithm A and two computable functions $f, g : N \to \mathrm{N}$ such that, given $(x, k) \in \Sigma^* \times N$, the algorithm A correctly decides whether $(x, k) \in L$ in time bounded by $f(k)|(x, k)|^{g(k)}$. The complexity class containing all slice-wise polynomial problems is called XP.

To rule out certain problems are not FPT, there is a notion of lower bound which is similar to the NP-completeness theory of polynomial time computation. We observe one difference though, there are different levels of hardness classes W[1],W[2],.. in parameterized complexity, unlike in classical complexity where all the NP hard problems are reducible to each other.

The primary assumption here is $FPT \neq W[1]$ which is a stronger assumption than $P \neq NP$. We introduce a notion of reduction to classify problems into such classes. If we can reduce a parameterized problem A to a parameterized problem B such that if B has an algorithm of a particular kind then so does A.

For our purposes, we mainly try to rule out the existence of an FPT algorithm for MkU problem. It is known that CLIQUE parameterized by solution size is $W[1]$ complete. This means that $W[1]$ is the set of all problems that can be obtained through a parameterized reduction from CLIQUE parameterized by solution size. We now recall the notion of parameterized reduction. If we can find a parameterized reduction from CLIQUE or some other problem X , then we can say that X cannot have an FPT algorithm unless $FPT = W[1]$.

Definition 0.0.4. *[3]* Let $A, B \subseteq \Sigma^* \times N$ be two parameterized problems. A *parameterized reduction* from A to B is an algorithm that, given an instance (x, k) of A, outputs an instance (x', k') of B such that

1. (x, k) is a yes-instance of A if and only if (x', k') is a yes-instance of B,
2. $k' \leq g(k)$ for some computable function g, and
3. the running time is $f(k)|x|^{O(1)}$ for some computable function f.

The following results hold for a parameterized reduction.

Theorem 0.0.1. *[3] If there is a parameterized reduction from A to B and B is FPT, then A is FPT as well.*

Proof. Let (x, k) be the instance of A and there is a parameterized reduction from A to B giving equivalent instance $(x^{'}, k^{'})$. As discussed above, the running time of it would be $f(k)|x|^{c_1}$, where c_1 is some constant. By definition of parameterized reduction, $k^{'} \leq g(k)$ and $|x^{'}| \leq f(k)|x|^{c_1}$ as running time of reduction should be an upper bound on size of produced instance. Now, B is FPT hence the reduced instance is solvable in time $h(k^{'})|x^{'}|^{c_2}$. By using relations mentioned above we get, $h(k^{'})|x^{'}|^{c_2} \leq h(g(k))|f(k)|x^{'}|^{c_1}|^{c_2}$. So total running time to solve A is equal to the time for reduction plus $h(g(k))|f(k)|x^{'}|^{c_1}|^{c_2} = f(k)|x|^{c_1} + h(g(k))|f(k)|x^{'}|^{c_1}|^{c_2} = f^{'}(k)|x|^{c_1 c_2}$, where $f^{'}(k) = h(g(k))f(k) + f(k)$ which is a computable function. Therefore A is FPT. □

Theorem 0.0.2. *[3] If there are parameterized reductions from A to B and from B to C, then there is a parameterized reduction from A to C.*

Proof. Let (x, k) be the instance of A and (x_1, k_1) be the instance of B reduced from A. Also, let (x_2, k_2) be the instance of C reduced from instance (x_1, k_1) of B. Now, let's suppose that we have parameterized reduction from A to B and B to C. For parameterized reduction from A to B we get, $k_1 \leq g_1(k)$ and $f_1(k)|x|^{c_1}$ to be the running time of reduction. Similarly for reduction from B to C, we get, $k_2 \leq g_2(k_1)$ and $f_2(k_1)|x|^{c_2}$. Here f_1, f_2, g_1, g_2 are all computable functions. Now, from above equations we can see that $k_2 \leq g_2(g_1(k))$ and reduction from A to C will have time complexity $g_2(f_1(k))(g_1(k)|x|^{c_1})^{c_2} = g_3(k)|x|^{c_3}$, where $g_3(k) = g_2(f_1(k))(g_1(k))^{c_2}$ and $c_3 = c_1 * c_2$. Now, (x, k) is yes instance of A if and only if (x_1, k_1) is an yes instance of B; and (x_1, k_1) is yes instance of B if and only if (x_2, k_2) is yes instance of C. Hence (x, k) is yes instance of A if and only if (x_2, k_2) is yes instance of C. Therefore reduction from A to C satisfies all the requirements of parameterized reduction. $\square$

Chapter 1

Preliminaries

We begin with the definition of tree decomposition of a given graph G. The goal is to provide a dynamic programming algorithm on a tree decomposition that finds a subset $S \subseteq V$ of size s having minimum size neighbourhood.

Definition 1.0.1. A *tree decomposition* of a graph G is a pair $(T, \{X_t\}_{t \in V(T)})$ where T is a tree and each node t of the tree T contains a bag $X_t \subseteq V(G)$, such that the following conditions are satisfied:

1. Each vertex of G is contained in at least one bag.
2. For every edge $uv \in E(G)$, both u and v are contained in at least one bag.
3. For every $u \in V(G)$, the set $\{t \in V(T) \mid u \in X_t\}$ induces a connected subtree of the tree T.

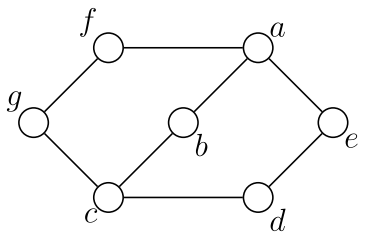
Definition 1.0.2. The *width* of a tree decomposition is defined as $width(T) = max_{t \in V(T)}|X_t| - 1$ and the treewidth $tw(G)$ of a graph G is the minimum width among all possible tree decompositions of G.

Definition 1.0.3. A tree decomposition $(T, \{X_t\}_{t \in V(T)})$ is said to be *nice tree decomposition* if the following conditions are satisfied:

1. All bags correspond to leaves are empty. One of the leaves is considered as root node r. Thus $X_r = \emptyset$ and $X_l = \emptyset$ for each leaf l.

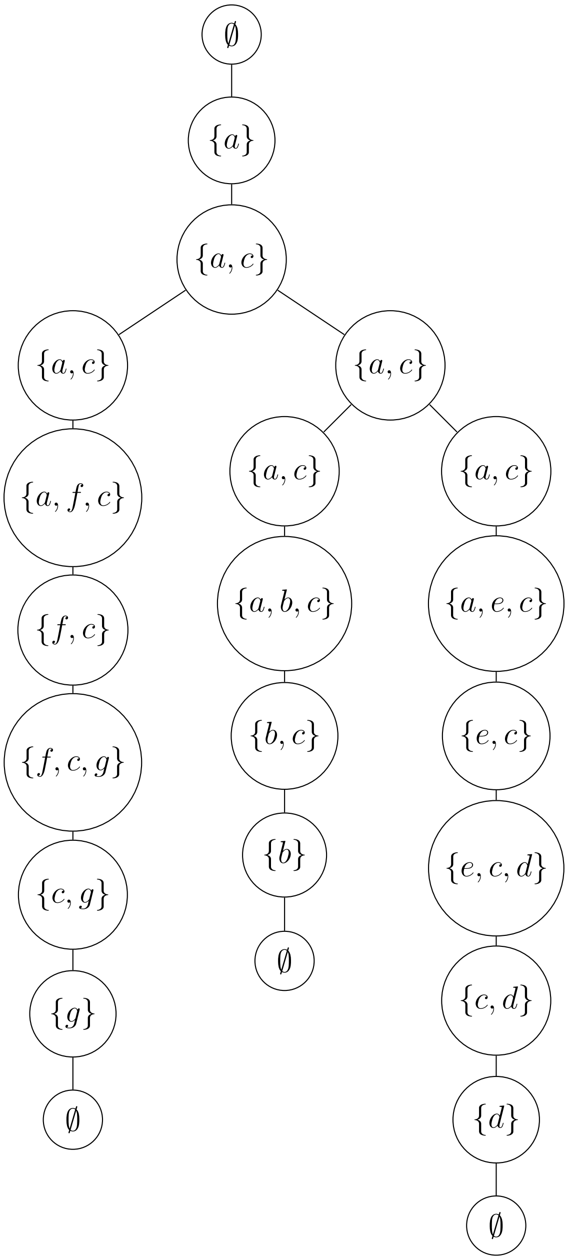
2. There are three types of non-leaf nodes:
   - Introduce node: we say a vertex v is introduced at node t if $X_t = X_{t'} \cup \{v\}$, where $v \notin X_{t'}$ and $t^{'}$ is the only child of t in T; we say node t is an *introduce node* and introducing vertex v.
   - Forget node: a node t is a *forget node* and forgetting vertex v if $X_t = X_{t'} \setminus \{v\}$, where $v \in X_{t'}$ and $t^{'}$ is the only child of t.
   - Join node: a node t is a join node if $X_t = X_{t_1} = X_{t_2}$, where t_1 and t_2 are two children of t.

Note that, by the third property of tree decomposition, a vertex $v \in V(G)$ may be introduced several time, but each vertex is forgotten only once. To control introduction of edges, sometimes one more type of node is considered in nice tree decomposition called introduce edge node. An introduce edge node is a node t, labeled with edge $uv \in E(G)$, such that $u, v \in X_t$ and $X_t = X_{t'}$, where $t^{'}$ is the only child of t. We say that node t introduces edge uv. Node t is inserted in nice tree decomposition as a child of forget node of u, given that u is forgotten before v.

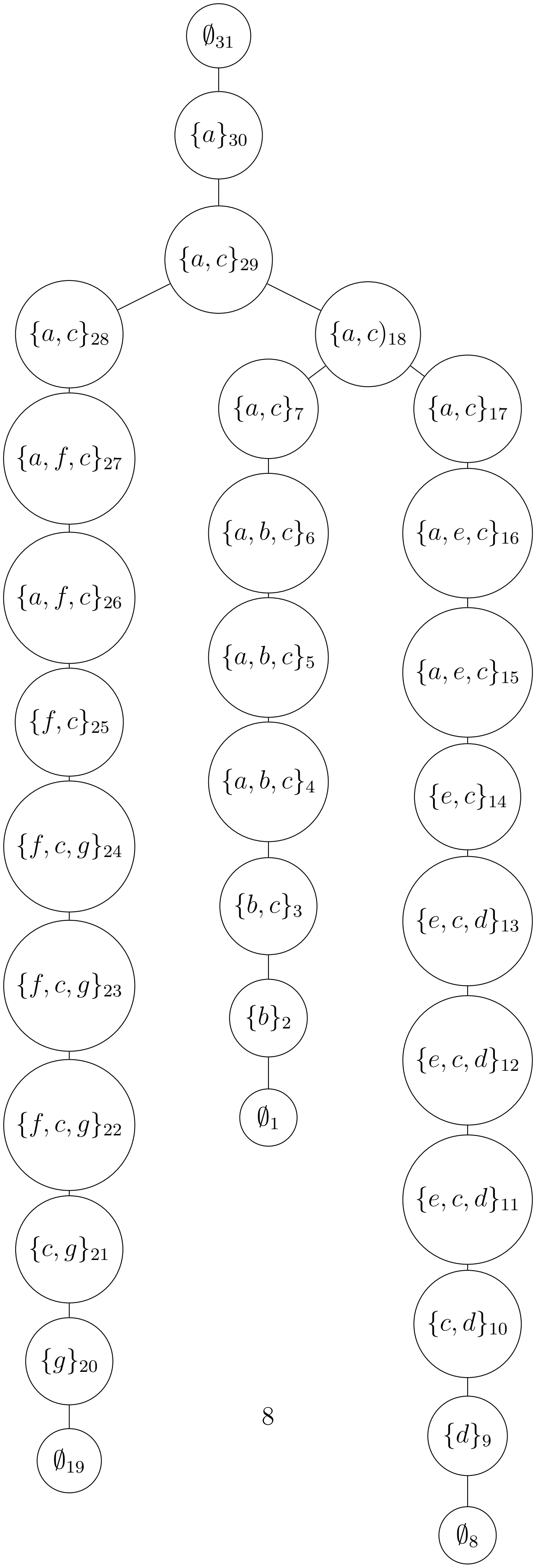


Let this graph be H

Nice tree decomposition for the graph H.



Nice tree decomposition with Introduces Edge Nodes is given below:
Nodes $5, 6, 12, 13, 16, 23, 24, 27$ are introduce edge nodes of edges $ab, bc, ed, dc, ae, fg, gc, af$ respectively.


$\emptyset_{31}$
$\{a\}_{30}$
$\{a, c\}_{29}$
$\{a, c\}_{28}$
$\{a, c)_{18}$
$\{a, c\}_{7}$
$\{a, c\}_{17}$
$\{a, f, c\}_{27}$
$\{a, b, c\}_{6}$
$\{a, e, c\}_{16}$
$\{a, f, c\}_{26}$
$\{a, b, c\}_{5}$
$\{a, e, c\}_{15}$
$\{f, c\}_{25}$
$\{a, b, c\}_{4}$
$\{e, c\}_{14}$
$\{f, c, g\}_{24}$
$\{b, c\}_{3}$
$\{e, c, d\}_{13}$
$\{f, c, g\}_{23}$
$\{b\}_{2}$
$\{e, c, d\}_{12}$
$\{f, c, g\}_{22}$
$\emptyset_{1}$
$\{e, c, d\}_{11}$
$\{c, g\}_{21}$
$\{c, d\}_{10}$
$\{g\}_{20}$
$\{d\}_{9}$
$\emptyset_{19}$
$\emptyset_{8}$

Lemma 1.0.1. *[3] A graph G with a tree decomposition of width at most k also has a nice tree decomposition of width at most k. Moreover, given a tree decomposition $(T, \{X_t\}_{t \in V(T)})$ of G of width at most k, its nice tree decomposition of width at most k that has at most $O(k|V(G)|)$ nodes can be computed in time $O(k^2. \max\{|V(G)|, |V(T)|\})$.*

1.1 Weighted Independent Set

In this section, we give an example of FPT dynamic programming algorithm using treewidth as a parameter. We will focus on weighted independent set problem. Given a graph G, where each vertex is assigned a weight, the task is to find weighted independent set of maximum weight in the graph. This is the maximum weighted independent set problem.

Let G be an n-vertex weighted graph and $(T, \{Y_i\}_{i \in V(T)})$ be the tree decomposition on G. We can assume that this is a nice tree decomposition using above lemma. Let r be the root node and let V_i be the union of all bags in subtree rooted at i including Y_i.

We will be defining a subproblem as finding maximum weighted independent $Z^{'}$, given $Z \subseteq Y_i$ and $Z \subseteq Z^{'}$ such that $Z^{'} \subseteq V_i$ and $Z^{'} \cap Y_i = Z$. We denote maximum possible weight of $Z^{'} = P[i, Z]$. We put $P[t, Z] = -\infty$ in case no such $Z^{'}$ exists. Our aim would be to find value of $P[r, \phi]$.

Now we will give recursive formulas:

Let S be any subset of Y_i and its independent, if not, then $P[i, Z] = -\infty$.

Leaf Node: If i is a leaf node then $P[i, \phi] = 0$.

Introduce vertex Node: If i is introduce vertex node with $i^{'}$ as a child then we know that $Y_i = Y_{i'} \cup \{m\}$, where m is the introduced vertex. Then following relation holds:

$$P[i, Z] = \begin{cases} P[i^{'}, Z] & \text{if } m \notin Z \\ P[i^{'}, Z \setminus \{m\}] + w(m) & \text{otherwise} \end{cases},$$

where $w(m)$ is weight of m.

Case 1: $m \notin Z$. Then all families of set $Z^{'}$ under consideration in $P[i, Z]$ and $P[i^{'}, Z]$ are equal, hence $P[i, Z] = P[i^{'}, Z]$

Case 2: $m \in S$. Assume $Z^{'}$ is maximum independent set attained in definition of $P[i, Z]$. Clearly $Z^{'} \setminus \{m\}$ comes under definition of $P[i^{'}, Z \setminus \{m\}]$, so we get $P[i^{'}, Z \setminus \{m\}] \geq w(Z^{'} \setminus \{m\}) = w(Z^{'}) - w(m) = P[i, Z] - w(m)$, which implies that $P[i, Z] \geq P[i^{'}, Z \setminus \{m\}] +$

$w(m)$. Conversely, let the maximum achieved in definition of $P[i', Z \setminus \{m\}]$ is Z^1, then $Z^1 \cap Y_{i'} = Z \setminus \{m\}$ and m does not a neighbour in $m_{i'} \setminus Y_{i'}$ so m does not have neighbour in $Z^1 \setminus Y_{i'}$. Hence, $Z^1 \cup \{m\}$ is independent set and comes in definition of $P[i, Z]$. So we get, $P[i, Z] \geq w(Z^1 \cup \{m\}) = w(Z^1) + w(m) = P[i', Z \setminus \{m\}] + w(m)$.

Combining two inequalities we get, $P[i, Z] = P[i', Z \setminus \{m\}] + w(m)$.

Forget Node: If i is a forget node with child i' then $Y_i = Y_{i'} \setminus \{v\}$, where v is the forgotten vertex. Then following relation holds:

$$P[i, Z] = max\{P[i', Z], P[i', Z \cup \{v\}]\}.$$

Proof for this formula is as below. Let Z' is maximum achieved in definition of $P[t, Z]$. If $v \notin Z$ then Z' comes under definition of $P[i', Z]$, which implies $P[i', Z] \geq w(Z') = P[i, Z]$. On the other hand if $v \in Z$ then Z' is considered in definition of $P[i', Z \cup \{v\}]$. So we get $P[i, Z] \leq max\{P[i', Z], P[i', Z \cup \{v\}]\}$.

As $P[i', Z]$ and $P[i', Z \cup \{v\}]$ are considered in definition of $P[i, Z]$, we get $P[i, Z] \geq P[i', Z]$ and $P[i, Z] \geq P[i', Z \cup \{v\}]$, which implies $P[i, Z] \geq max\{P[i', Z], P[i', Z \cup \{v\}]\}$.

Combining both inequalities we get the recursive formula.

Join Node: If i is a join node with i_1 and i_2 as its children then $Y_i = Y_{i_1} = Y_{i_2}$. The recursive formula is

$$P[i, Z] = P[i_1, Z] + P[i_2, Z] - w(Z)$$

The proof is as follows. Let Z' be the maximum set in definition of $P[i, Z]$ and $S_1 = Z' \cap V_{i_1}, Z_2 = Z' \cap V_{i_2}$. Then we can see that S_1 is independent and $S_1 \cap Y_{i_1} = Z$, so it comes under definition of $P[i_1, Z]$, hence we have $P[i_1, Z] \geq w(Z_1)$. Similarly we have $P[i_2, Z] \geq w(Z_2)$. Since $Z_1 \cap Z_2 = Z$, we get, $P[i, Z] = w(Z') = w(Z_1) + w(Z_2) - w(Z) \leq P[i_1, Z] + P[i, Z] - w(Z)$.

Conversely, let Z_1' be the maximum achieved in definition of $P[i_1, Z]$ and Z' in $P[i_2, Z]$. Now we know that there is no edge between vertices of $V_{i_1} \setminus Y_i$ and $V_{i_2} \setminus Y_i$, therefore $Z_3 = Z_1 \cup Z_2$ is independent and we have $Z_3 \cap Y_i = Z$, which means Z_3 is in definition of $P[i, Z]$. Hence $P[i, Z] \geq w(Z_3) = w(Z_1) + w(Z_2) - w(Z) = P[i_1, Z] + P[i_2, Z] - w(Z)$.

Combining two inequalities we get the recursive formula.

We can compute each value $P[i, Z]$ in time $k^{O(1)}$ and number of subsets Z of Y_i is 2^k. So to compute all the values of $P[i, Z]$ for each i will require $2^k k^{O(1)}$ time. As there are $O(kn)$ nodes in tree decomposition total time required is $2^k k^{O(1)} n$.

Chapter 2

NP-completeness of the minimum neighbourhood problem

In this chapter, we prove that minimum neighbourhood problem is NP-complete. Here is the decision version of the minimum neighbourhood problem. We are given a graph $G = (V, E)$ with n vertices and two positive integers $k \leq n$ and ℓ. Does G contain a set $S \subseteq V$ of size k such that $|N_G[S]| \leq \ell$? Now we state the decision version of Minimum k-Union (MkU) problem.

Definition 2.0.1. *In MkU problem, we are given an universe* $U = \{1, 2, \ldots, n\}$ *of* n *elements and a collection of sets* $\mathcal{S} \subseteq 2^U$*, as well as two integers* $k \leq |\mathcal{S}|$ *and* ℓ*. Does there exist a collection* $T \subseteq \mathcal{S}$ *with* $|T| = k$ *such that* $|\cup_{S \in T} S| \leq \ell$*.*

Theorem 2.0.1. *The MkU problem is NP-hard.*

Now we prove that the minimum neighbourhood problem is NP-complete.

Theorem 2.0.2. *The minimum neighbourhood problem is NP-complete.*

Proof. We first show that minimum neighbourhood problem is in NP. Given a graph $G = (V, E)$ with n vertices and two integers $k \leq n$ and ℓ, a certificate could be a set $S \subseteq V$ of size k. We could then check, in polynomial time, there are k vertices in S, and the size of $N_G[S]$ is less than or equal to ℓ.

We prove the minimum neighbourhood problem is NP-hard by showing that that Minimum k-Union problem $\leq_P$ Minimum Neighbourhood Problem. Given an instance $(U, \mathcal{S}, k, \ell)$ of MkU problem, we construct a bipartite graph H with bipartition X and Y. The vertices in $X = \{u_1, u_2, \ldots, u_n\}$ are the elements in U; the vertices in $Y = \{s_1, s_2, \ldots, s_m\}$ correspond to sets in $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$. We make $u_i \in X$ adjacent to $s_j \in Y$ if and only if $u_i \in S_j$. Additionally, for each vertex u_i, we add a clique of size $n+1$, K^i_{n+1} and we make u_i adjacent to each vertex in K^i_{n+1}.

We show that there is a collection of k sets $\left\{S_{i_1}, S_{i_2}, \ldots, S_{i_k}\right\} \subseteq \mathcal{S}$ such that $|\cup_{j=1}^{k} S_{i_j}| \leq \ell$, for Minimum k-Union problem if and only if there is a set $S \subseteq V(H)$ of k vertices such that $|N_H[S]| \leq k+\ell$, for Minimum Neighbourhood Problem. Suppose there is a collection of k sets $\left\{S_{i_1}, S_{i_2}, \ldots, S_{i_k}\right\} \subseteq \mathcal{S}$ such that $|\cup_{j=1}^{k} S_{i_j}| \leq \ell$. We choose the vertices $\{s_{i_1}, s_{i_2}, \ldots, s_{i_k}\} \subseteq Y$ correspond to sets $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$. As the size of the union of these k sets $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$ is less or equal to ℓ, the closed neighbourhood of $s_{i_1}, s_{i_2}, \ldots, s_{i_k}$ will contain $s_{i_1}, s_{i_2}, \ldots, s_{i_k}$ and at most ℓ vertices u, where $u \in \cup_{j=1}^{k} S_{i_j}$. Hence the size of the closed neighbourhood of $s_{i_1}, s_{i_2}, \ldots, s_{i_k}$ is at most $k + \ell$.

Conversely, suppose there is a collection $S \subseteq V(H)$ of k vertices that has a closed neighbourhood of size at most $k + \ell < n$. S cannot contain any vertex from X as each vertex in X has at least $n + 2$ closed neighbour in H. S cannot contain any vertex from K^i_{n+1}, as each vertex in K^i_{n+1} has $n + 2$ closed neighbours in H. Thus $S \subseteq Y$ and let $S = \{s_{j_1}, s_{j_2}, \ldots, s_{j_k}\}$. We consider the k sets $S_{j_1}, S_{j_2}, \ldots, S_{j_k}$ correspond to these k vertices in S. As S has at most $k+\ell$ closed neighbours, $|\cup_{i=1}^{k} S_{j_i}| \leq \ell$. This completes the proof. $\square$

Chapter 3

Minimum Neighbourhood Problem

In this chapter, we propose a dynamic programming algorithm for minimum neighbourhood problem. Recall that given a graph $G = (V, E)$ and a positive integer p, we want to find $S \subseteq V$ such that $|S| = p$ and the size of $N[S]$ is minimum. We provide a dynamic programming algorithm on a tree decomposition of G. Given a graph G, an integer p and a tree decomposion $(T, X_t : t \in V(T))$, subproblems will be defined on $G_t = (V_t, E_t)$ where V_t is the union of all bags present in subtree of T rooted at t, including X_t and E_t is the set of edges e introduced in the subtree rooted at t. We define a colour function $f : \ X_t \mapsto \{0, 1, \hat{0}, \hat{1}\}$ that assigns four different colours to the vertices of X_t. The meanings of four different colour are given below:

1 (black vertices): vertices contained in set S whose neighbourhood size we wish to calculate in G_t.
0 (white vertices): vertices adjacent to black vertices, these vertices are contained in partial solution in G_t.
$\hat{0}$ (green vertices): vertices not adjacent to black vertices in G_t.
$\hat{1}$ (gray vertices): vertices whose colour (black, white or green) has not been decided yet.

At the end of algorithm, the vertices of G will be coloured by colours black, white and green, no vertex will be of grey colour, that is no vertex will be left undecided. The reason behind using grey colour is that some vertices of a bag may be in S or in $N(S)$ depending on the vertices and edges which are not introduced so far. So we consider subproblems where role of some vertices are left undecided, since such subproblems are important for getting the optimal solution. Now we introduce some notations. Let $X \subseteq V$ and consider a colouring $f : X \mapsto \{1, 0, \hat{0}, \hat{1}\}$. For $\alpha \in \{1, 0, \hat{0}, \hat{1}\}$ and $v \in V(G)$ a new colouring $f_{v \mapsto \alpha} : X \cup \{v\} \mapsto \{1, 0, \hat{0}, \hat{1}\}$ is defined as follows:

$$f_{v \mapsto \alpha}(x) = \begin{cases} f(x) & \text{when } x \neq v \\ \alpha & \text{when } x = v \end{cases}$$

Let f be a colouring of X, then the notation $f_{|Y}$ is used to denote the restriction of f to Y, where $Y \subseteq X$.

For a colouring f of X_t, we denote by $c[t, f, i]$ the minimum size of $N(S) \subseteq V_t$ such that

1. $S \subseteq V_t$ and $|S| = i$.
2. $S \cap X_t = f^{-1}(1)$ which is the set of vertices of X_t coloured black.
3. $N(S) \cap X_t = f^{-1}(0)$, which is the set of vertices of X_t coloured white.
4. Each vertex in $V_t \setminus f^{-1}(\hat{1})$ is either in S, $N(S)$ or non-adjacent in G_t to the vertices in set S. As all grey$(\hat{1})$ vertices belong to X_t, removal of $f^{-1}(\hat{1})$ from X_t will result in removal of all grey$(\hat{1})$ vertices from V_t.

We call such a set $N(S)$ a *minimum neighbourhood set compatible* for (t, f, i). If no compatible $N[S] \setminus S$ exists, then we put $c[t, f, i] = \infty$ also $c[t, f, i < 0] = \infty$. Since each vertex in X_t can be coloured with 4 colours $(1, 0, \hat{0}, \hat{1})$, the number of possible colourings f of X_t is $4^{|X_t|}$ and for each colouring f we vary i from 0 to p. The size of minimum neighbourhood $N[S] \setminus S$ of G with $|S| = p$ will be $c[r, \phi, p]$, where r is the root node of tree decomposition of G as $G = G_r$ and $X_r = \emptyset$. Now we present the recursive formulae for the values of c.

Leaf node: If t is a leaf node, then the corresponding bag X_t is empty. Hence the colour function f on X_t is an empty colouring; the number i of vertices coloured black cannot be greater than zero. Thus we have $c[t, \emptyset, i = 0] = 0$ and $c[t, \emptyset, i > 0] = \infty$.

Introduce node: Suppose t is an introduce node with child t' such that $X_t = X_{t'} \cup \{v\}$ for some $v \notin X_{t'}$. The introduce node introduces the vertex v but does not introduce the edges incident to v to G_t. So when v is introduced by node t it is an isolated vertex in G_t. Vertex v cannot be coloured white 0; as it is isolated and it cannot be neighbour of any black vertex. Hence if $f(v) = 0$, then $c[t, f, i] = \infty$. When $f(v) = 1$, v is contained in S. As v is an isolated vertex, it does not contribute towards the size of $N(S)$, hence $c[t, f, i] = c[t', f_{|X_{t'}}, i - 1]$. When $f(v) = \hat{0}$ or $f(v) = \hat{1}$, v does not contribute towards the size of $N(S)$. Here minimum neighbourhood set compatible for $(t^{'}, f_{|X_{t'}}, i)$ is the same as minimum neighbourhood set compatible for (t, f, i). So, $c[t, f, i] = c[t^{'}, f_{|X_{t'}}, i]$. Combining all the cases together, we get

$$c[t, f, i] = \begin{cases} \infty & \text{if } f(v) = 0 \\ c[t^{'}, f_{|X_{t'}}, i - 1] & \text{if } f(v) = 1 \\ c[t^{'}, f_{|X_{t'}}, i] & \text{otherwise} \end{cases}$$

Introduce edge node: Let t be an introduce edge node that introduces the edge (u, v), let t' be the child of t. Thus $X_t = X_{t'}$; the edge (u, v) is not there in t', but it is there in t. Let f be a colouring of X_t. We consider the following cases:

- Suppose $f(u) = 1$ and $f(v) = \hat{0}$. This means $u \in S$ and v is non-adjacent to black vertices in G_t. But u and v are adjacent in G_t. Thus $c[t, f, i] = \infty$. The same conclusion can be drawn when v is coloured black and u is coloured green.

- Suppose $f(u) = 1$ and $f(v) = 0$. This means $u \in S$ and $v \in N(S)$ in G_t. In order to get a minimum neighbourhood set compatible for (t, f, i), we consider precomputed solution for t' where the colour of v is grey, that is , we consider precomputed minimum neighbourhood set compatible for $(t', f_{v \to \hat{1}}, i)$. The size of minimum neighbourhood compatible set for (t, f, i) is one more than the size of minimum neighbourhood compatible set for $(t', f_{v \to \hat{1}}, i)$, that is, $c[t, f, i] = 1 + c[t^{'}, f_{u \mapsto \hat{1}}, i]$. The same conclusion can be drawn when v is coloured black and u is coloured white.

- Other colour combinations of u and v do not affect the size of $N(S)$ or do not contradict the definition of campatability. So minimum neighbourhood set compatible for $t^{'}, f_{|X_{t'}}, i$ is the same as minimum neighbourhood compatible set for t, f, i and hence $c[t, f, i] = c[t^{'}, f_{|X_{t'}}, i]$.

Combining all the cases together, we get

$$c[t,f,i] = \begin{cases} \infty & \text{if } [f(u),f(v)] = [\hat{0},1] \\ \infty & \text{if } [f(u),f(v)] = [1,\hat{0}] \\ c[t^{'},f_{v\mapsto\hat{1}},i]+1 & \text{if } [f(u),f(v)] = [1,0] \\ c[t^{'},f_{u\mapsto\hat{1}},i]+1 & \text{if } [f(u),f(v)] = [0,1] \\ c[t^{'},f_{|X_{t^{'}}},i] & \text{otherwise} \end{cases}$$

Forget node: Let t be a forget node with the child t' such that $X_t = X_{t'} \setminus \{w\}$ for some vertex $w \in X_{t'}$. Here the bag X_t forgets the vertex w. At this stage we decides the final colour of the vertex w. We observe that $G_{t'} = G_t$. The closed neighbourhood sets compatible for $(t^{'}, f_{w\mapsto 1}, i)$, $(t^{'}, f_{w\mapsto 0}, i)$, $(t^{'}, f_{w\mapsto\hat{0}}, i)$ are also compatible for (t, f, i). On the other hand the closed neighbourhood compatible set for (t, f, i) is also compatible for $(t^{'}, f_{w\mapsto 1}, i)$ if $w \in S$ or $(t^{'}, f_{w\mapsto 0}, i)$ if $w \in (N[S] \setminus S)$ or $(t^{'}, f_{w\mapsto\hat{0}}, i)$ if $w \notin N[S]$. Hence

$$c[t,f,i] = \min\left\{c[t^{'},f_{w\mapsto 1},i], c[t^{'},f_{w\mapsto 0},i], c[t^{'},f_{w\mapsto\hat{0}},i]\right\}$$

Join Node: Let t be a join node with children t_1 and t_2, such that $X_t = X_{t_1} = X_{t_2}$. Let f be a colouring of X_t. We say that colourings f_1 of X_{t_1} and f_2 of X_{t_2} are consistent for colouring f of X_t, if the following conditions are true for each $v \in X_t$:

1. $f(v) = 1$ if and only if $f_1(v) = f_2(v) = 1$
2. $f(v) = \hat{0}$ if and only if $f_1(v) = f_2(v) = \hat{0}$
3. $f(v) = \hat{1}$ if and only if $f_1(v) = f_2(v) = \hat{1}$
4. $f(v) = 0$ if and only if $(f_1(v), f_2(v)) = (0, \hat{1})$ or $(\hat{1}, 0)$

Let f ba a colouring of X_t; f_1 and f_2 be two colourings of X_{t_1} and X_{t_2} respectively consistent with f. Suppose $N[S_1] \setminus S_1$ is a neighbourhood compatible set for (t_1, f_1, i_1) and $N[S_2] \setminus S_2$ is a neighbourhood compatible set for (t_2, f_2, i_2), where $|S_1| = i_1$ and $|S_2| = i_2$. Set $S = S_1 \cup S_2$, clearly $|S| = |S_1| + |S_2| - |f^{-1}(1)|$. It is easy to see that $N[S] \setminus S = (N[S_1] \setminus S_1) \cup (N[S_2] \setminus S_2)$ is a neighbourhood compatible set for (t, f, i), where $i = i_1 + i_2 - |f^{-1}(1)|$. According to Condition 4 in the definition of consistent function, each $v \in X_t$ that is white in f, we make it white either in f_1 or f_2. In other words, for such S_1 and S_2, we have $(N[S_1] \setminus S_1) \cap (N[S_2] \setminus S_2) = \emptyset$; it follows that

$$|N[S] \setminus S| = |(N[S_1] \setminus S_1)| + |(N[S_2] \setminus S_2)|.$$

Consequently, we have the following recursive formula:

$$c(t, f, i) = \min_{f_1, f_2} \left\{ \min_{i_1, i_2 \,:\, i = i_1 + i_2 - |f^{-1}(1)|} \Big\{ c(t_1, f_1, i_1) + c(t_2, f_2, i_2) \Big\} \right\}.$$

We now analyse the running time of the algorithm. The time needed to process each leaf node, introduce vertex node, introduce edge node or forget node is $4^k k^{O(1)} p$ as each bag X_t can be coloured in 4^k ways, adjacency of vertices can be checked in $k^{O(1)}$ time and for each colouring f we vary i from 0 to p, where k is tree width and hence $|X_t| \leq k$. The computation of c value for join node takes more time and it can be done as follows. If colourings f_1 and f_2 are consistent with f, then for every $v \in X_t$ we have $(f(v), f_1(v), f_1(v)) \in \{(1,1,1), (\hat{0},\hat{0},\hat{0}), (\hat{1},\hat{1},\hat{1}), (0,0,\hat{1}), (0,\hat{1},0)\}$. Hence there are exactly $5^{|X_t|}$ triples of colourings (f, f_1, f_2) such that f_1 an f_2 are consistent with f, since we have 5 possibilities of $(f(v), f_1(v), f_2(v))$ for every vertex $v \in X_t$. In order to compute $c(t, f, i)$, we iterate through all triples (f, f_1, f_2); then for each considered triple (f, f_1, f_2) we vary i_1 from 0 to p and i_2 varies according to equation $i = i_1 + i_2 - |f^{-1}|$. Also i varies from 0 to p. So the time needed for each join node is $5^k k^{O(1)} p^2$. There are $O(kn)$ nodes in a nice tree decomposition. Therefore, the time complexity of the algorithm is $5^k k^{O(1)} p^2 n$, where $n = |V(G)|$.

Chapter 4

Conclusions

Given a graph $G = (V, E)$ and an integer k, we want to find a $S \subset V$, such that $|S| = k$ and the cardinality of $N[S]$ is minimum. This problem is called minimum neighbourhood problem. We propose a fix parameter tractable (FPT) algorithm for minimum neighbourhood problem parameterized by the treewidth of the graph G. It is an interesting open problem to study minimum neighbourhood problem with respect to the other parameters. There is no known FPT algorithm for minimum neighbourhood problem when parameterized with respect to the solution size. It is also interesting to study parameterized complexity of minimum neighbourhood problem for special graph classes like, chordal graph, interval graphs, proper interval graphs, split graphs, etc.

Appendices

Appendix A

Python Code

This is a python code for the dynamic programming algorithm discussed above, using graph H and its nice tree decomposition with introduce edge nodes as an input (from page 8).

```
#class vercol is defined to assign colour to vertices
class vercol:
    def __init__(whose, colour):
        whose.colour = colour

#All vertices are assigned grey colour. This will be the default
#colour of vertices. As program progresses, their colour will change
#according to algorithm.
a = vercol("grey")
b = vercol("grey")
c = vercol("grey")
d = vercol("grey")
e = vercol("grey")
f = vercol("grey")
g = vercol("grey")


#m is the total number of vertices in the input graph.
m=7


#class bag is defined to create nodes and assign properties to them.
class bag:
```

```
        def __init__(whose, vertices, children1, nodetype, herovertex,
        number):
            whose.vertices = vertices
            whose.children1 = children1
            whose.type = nodetype
            whose.hero = herovertex
            whose.number = number

    #1st entry assigns vertices to node, 2nd is the child of node which
    #establishes connection between current node to its child node,
    #3rd entry assigns node type (LN = leaf node, IVN = introduce vertex
    #node, IEN = introduce edge node, FN = forget node, JN = join node),
    #4th entry (herovertex) assigns that vertex to the node which
    #defines its node type. For example bag2 is introduce vertex node of
    #vertex b (hence called herovertex),
    #5th entry is the number assigned to the bag.
    bag1  = bag([], [], "LN", [], 1)
    bag2  = bag([b], bag1, "IVN", [b], 2)
    bag3  = bag([b, c], bag2, "IVN", [c], 3)
    bag4  = bag([a, b, c], bag3, "IVN", [a], 4)
    bag5  = bag([a, b, c], bag4, "IEN", [a, b], 5)
    bag6  = bag([a, b, c], bag5, "IEN", [b, c], 6)
    bag7  = bag([a, c], bag6, "FN", [b], 7)
    bag8  = bag([], [], "LN", [], 8)
    bag9  = bag([d], bag8, "IVN", [d], 9)
    bag10 = bag([c, d], bag9, "IVN", [c], 10)
    bag11 = bag([e, c, d], bag10, "IVN", [e], 11)
    bag12 = bag([e, c, d], bag11, "IEN", [e, d], 12)
    bag13 = bag([e, c, d], bag12, "IEN", [d, c], 13)
    bag14 = bag([e, c], bag13, "FN", [d], 14)
    bag15 = bag([a, e, c], bag14, "IVN", [a], 15)
    bag16 = bag([a, e, c], bag15, "IEN", [a, e], 16)
    bag17 = bag([a, c], bag16, "FN", [e], 17)
    bag18 = bag([a, c], bag17, "JN", [], 18)
    bag19 = bag([], [], "LN", [], 19)
    bag20 = bag([g], bag19, "IVN", [g], 2)
    bag21 = bag([c, g], bag20, "IVN", [c], 21)
    bag22 = bag([f, c, g], bag21, "IVN", [f], 22)
    bag23 = bag([f, c, g], bag22, "IEN", [f, g], 23)
    bag24 = bag([f, c, g], bag23, "IEN", [g, c], 24)
    bag25 = bag([f, c], bag24, "FN", [g], 25)
```

```
    bag26 = bag([a, f, c], bag25, "IVN", [a], 26)
    bag27 = bag([a, f, c], bag26, "IEN", [a, f], 27)
    bag28 = bag([a, c], bag27, "FN", [f], 28)
    bag29 = bag([a, c], bag28, "JN", [], 29)
    bag30 = bag([a], bag29, "FN", [c], 30)
    bag31 = bag([], bag30, "FN", [a], 31)


    #parent() function defines the parental relation between nodes, so now the nodes are
    #connected to their parent nodes.
    def parent(x):
        if x==bag1:
            return bag2
        elif x==bag2:
            return bag3
        elif x==bag3:
            return bag4
        elif x==bag4:
            return bag5
        elif x==bag5:
            return bag6
        elif x==bag6:
            return bag7
        elif x==bag7:
            return bag18
        elif x==bag8:
            return bag9
        elif x==bag9:
            return bag10
        elif x==bag10:
            return bag11
        elif x==bag11:
            return bag12
        elif x==bag12:
            return bag13
        elif x==bag13:
            return bag14
        elif x==bag14:
            return bag15
        elif x==bag15:
```

```
        return bag16
    elif x==bag16:
        return bag17
    elif x==bag17:
        return bag18
    elif x==bag18:
        return bag29
    elif x==bag19:
        return bag20
    elif x==bag20:
        return bag21
    elif x==bag21:
        return bag22
    elif x==bag22:
        return bag23
    elif x==bag23:
        return bag24
    elif x==bag24:
        return bag25
    elif x==bag25:
        return bag26
    elif x==bag26:
        return bag27
    elif x==bag27:
        return bag28
    elif x==bag28:
        return bag29
    elif x==bag29:
        return bag30
    elif x==bag30:
        return bag31

#children2() defines the second child of node if it has any (join
#node has two children).
def children2(x):
    if x==bag18:
        return bag7
    elif x==bag29:
        return bag18

#This concludes input.
```

```


    #Creating a list of length n+2, where n is the total number of nodes.
    colourlist=[]
    n=31
    for i in range(n+1):
        colourlist.append(i)


    #Defining minfun function which embeds the recursive formula for
    #join node.
    #It takes s and u as input where s is a node and u is an integer.
    def minfun(s, u):

        #Empty lists are created.
        minlist=[]
        blacklist=[]
        whitelist=[]

        #This 'for' loop insures that all black vertices and all white
        #vertices in node s go into blacklist and whitelist
        #respectively.
        for x in s.vertices:
            if x.colour == "white":
                whitelist.append(x)
            elif x.colour == "black":
                blacklist.append(x)

        #Defining function r with c, v and q as inputs, where c is a
        #list, v is an integer and q is a node.
        #Function w will be defined later.
        #This function assigns colours from list c to the vertices in
        #whitelist and returns function w taking input as one of the
        #children of q as an input.
        def r(c, v, q):
            for x in range(len(whitelist)):
                whitelist[x].colour = c[x]
            return W(q.children1, v, q.hero)

        #Defining function rr with c, v and q as inputs, where c is a
        #list, v is an integer and q is a node.
```

```
        #Function w will be defined later.
        #This function assigns colours from list c to the vertices in
        #whitelist and returns function w taking input as other child of
        #q as an input.
        def rr(c, v, q):
            for x in range(len(whitelist)):
                whitelist[x].colour = c[x]
            return W(children2(q), v, q.hero)

        #n is assigned the value equal to length of whitelist created
        #earlier.
        #List cash is created whose each entry is a list. Each entry is
        #n length long list and its entry can either be "white" or
        #"grey".
        #List cash contains all such permutations of n length list with
        #"white" or "grey" as entries.
        #Length of cash will be 2^n.
        #Each entry of recash is complementery opposite to entry at the
        #same position in cash.
        #For example if an entry at 4th position in cash looks like
        #["white","grey"] then entry at 4th position in recash will be
        #["grey","white"].
        import itertools
        n=len(whitelist)
        cash = list(itertools.product(["white", "grey"], repeat=n))
        recash = cash[::-1]

        #For each entry in cash and for each t (from 0 to u+m+1) we
        #calculate p and append it to minlist.
        #Then minimum entry in minlist is returned.
        #This loop represents the recursive relation of join node.
        for x in cash:
            for t in range(u+m+1):
                p = r(x,t,s) + rr(recash[cash.index(x)],u-t+len(blacklist),s)
                minlist.append(p)
        return min(minlist)


    #colourlist will be used to keep the record of colour of all vertices at
    #each step.
    #Here nth element of colourlist is substituted with current colourings of
```

```
    #vertices. As at this step all vertices are ”grey” coloured.
    colourlist[n]=[a.colour, b.colour, c.colour, d.colour, e.colour, f.colour, g.colour]

    #Function W represents the recursive relations.
    #It takes node, z and herocolour as an input, where z is an integer and
    #herocolour is in a form of a string.
    #herocolour is the colourings assigned by the recurrence relation of
    #parent node to herovertex which are then used by child node.
    #z is the integer p which is the size of vertex set whose minimum
    #neighbourhood size we want to find out.
    def W(node, z, herocolour=[”grey”]):

            #Here vertices are coloured by the colourings bestowed upon
            #by their parent node which we already stored in colourlist,
            #except for join node.
            #Each time herovertex of parent node will be coloured in
            #different colour. So after each iteration, colouring given
            #by parent nodes to other vertices must be remembered.
            #But this is not the case with join node as join node does
            #not have a herovertex it has only one iteration in this
            #function (i.e, function W).
            #The iterations in recurcive relations of join node are
            #taken care of in minfun function and not in function W.
            if node.type==”JN”:
                None
            else:
                [a.colour, b.colour, c.colour, d.colour, e.colour, f.colour,
                g.colour]=colourlist[node.number]

            #Here herovertex is coloured as the recursive relation of
            #parent node commanded i.e, colour of herovertex is changed
            #to colours in herocolour list.
            #Again leaf node and children of join node will be excluded
            #from here as leaf node doesn't have a child and join node
            #does not have herovertex.
            if parent(node) == [] or parent(node).hero == []:
                None
            else:
                for k in range(len(herocolour)):
                    parent(node).hero[k].colour = herocolour[k]
```

```

        #This is recurrence relation for Introduce Vertex Node.
        if node.type=="IVN":
            if node.hero[0].colour=="white":
                return float('inf')
            elif node.hero[0].colour=="black":
                colourlist[node.children1.number]=[a.colour, b.colour,
                c.colour, d.colour, e.colour, f.colour, g.colour]
                return W(node.children1, z-1, ["black"])
            elif node.hero[0].colour=="green":
                colourlist[node.children1.number]=[a.colour, b.colour,
                c.colour, d.colour, e.colour, f.colour, g.colour]
                return W(node.children1, z, ["green"])
            else:
                colourlist[node.children1.number]=[a.colour, b.colour,
                c.colour, d.colour, e.colour, f.colour, g.colour]
                return W(node.children1, z)

        #This is recurrence relation for Forget Node.
        elif node.type=="FN":
            colourlist[node.children1.number]=[a.colour, b.colour,
            c.colour, d.colour, e.colour, f.colour, g.colour]
            return min( W(node.children1, z, ["black"]),
           W(node.children1, z, ["white"]), W(node.children1, z,
            ["green"]) )

        #This is recurrence relation for Introduce Edge Node.
        elif node.type=="IEN":
            if node.hero[0].colour=="black" and
            node.hero[1].colour=="green":
                return float('inf')
            elif node.hero[0].colour=="green" and
            node.hero[1].colour=="black":
                return float('inf')
            elif node.hero[0].colour=="black" and
            node.hero[1].colour=="white":
                colourlist[node.children1.number]=[a.colour, b.colour,
                c.colour, d.colour, e.colour, f.colour, g.colour]
                return W(node.children1, z, ["black","grey"]) + 1
            elif node.hero[0].colour=="white" and
            node.hero[1].colour=="black":
```

```
                colourlist[node.children1.number]=[a.colour, b.colour,
                c.colour, d.colour, e.colour, f.colour, g.colour]
                return W(node.children1, z, ["grey","black"]) + 1
            else:
                colourlist[node.children1.number]=[a.colour, b.colour,
                c.colour, d.colour, e.colour, f.colour, g.colour]
                return W(node.children1, z,
                [node.hero[0].colour,node.hero[1].colour])

        #This is recurrence relation for join node (by using minfun).
        elif node.type=="JN":
            colourlist[node.children1.number]=[a.colour, b.colour,
            c.colour, d.colour, e.colour, f.colour, g.colour]
            colourlist[children2(node).number]=[a.colour, b.colour,
            c.colour, d.colour, e.colour, f.colour, g.colour]
            return minfun(node, z)

        #This is the base case of algorithm.
        elif node.type=="LN":
            if z==0:
                return 0
            else:
                return float('inf')


#Finally we call function w with inputs as the root node which in
#this case is bag26 and parameter p(size of vertex set whose minimum
#neighbourhood size we are about to find) whose value is 8 in this
#particular case.
print(W(bag26, 8))

```

Listing A.1: Python example

Bibliography

[1] Bodlaender, H., On linear time minor tests with depth-first search, *J. Algorithms*, 14, pp. 1-23, 1993.

[2] Christos Papadimitriou, Sanjoy Dasgupta, and Umesh Vazirani, Algorithms, Mc Graw Hill, 2006.

[3] Cygan, M., Fomin, F.V., Kowalik, L., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., Saurabh, S., Parameterized Algorithms, Springer, 2015.

[4] Downey, Rodney G., Fellows, M.R., Parameterized Complexity, Springer, 1999.

[5] Eden Chlamtac, Michael Dinitz, Yury Makarychev. Minimizing the Union: Tight Approximations for Small Set Bipartite Vertex Expansion, 24 Nov. 2014.

[6] Chen, J., Kneis, J., Lu, S., Mlle, D., Richter, S., Rossmanith, P., Sze, S.H., Zhang, F., Randomized divide-and-conquer: improved path, matching, and packing algorithms, *SIAM J. Computing* 38(6), 2526-2547, 2009.

[7] Koutis, I., Faster algebraic algorithms for path and packing problems. In Proceedings of the 35th International Colloquium of Automata, Languages and Programming (ICALP), Lecture Notes in Comput. Sci., vol. 5125, pp. 575-586, 2008.

[8] Naor, M., Schulman, L.J., Srinivasan, A., Splitters and near-optimal derandomization. *In Proceedings of the 36th Annual Symposium on Foundations of Computer Science* (FOCS), pp. 182-191. IEEE, 1995.

[9] Philipp Zschoche, Till Fluschnik, Hendrik Molter, Rolf Niedermeier, Complexity of Finding Small Separators in Temporal Graphs, 2015.

[10] Seroussi, G. and Bshouty, N.H., Vector sets for exhaustive testing of logic circuits, *IEEE Transactions on Information Theory*, Vol 34(3), 513-522, 1988.