Parameterized Algorithms and Experiments

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Vikas Naik.D and M.A.Azeem
(111901052 and 111901032)
Mentor : Dr.Kritika Ramaswamy

INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY PALAKKAD

CERTIFICATE

*This is to certify that the work contained in the project entitled "**Parameterized Algorithms and Experiments**" is a bonafide work of **Mohammed Abdul Azeem** (**Roll No. 111901032**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my guidance and that it has not been submitted elsewhere for a degree.*

Dr. Krithika Ramaswamy

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Palakkad

Abstract

Many computational problems are NP-hard and we do not expect efficient algorithms for solving them. Parameterized complexity is one of the approaches to this hardness where one designs algorithms whose exponential factor in the running time is restricted to a parameter associated with the input. Informally, a parameterized algorithm is an efficient algorithm on instances where the parameter is small. A kernelization algorithm for a parameterized problem is a polynomial-time algorithm which transforms an arbitrary instance of the problem into an equivalent instance of the same problem whose size is bounded by a function of the parameter of the original instance.

In this work, we aim to study the efficacy of kernelization procedures on standard benchmarks instances (like BHOSLIB, DIMACS and instances created for PACE) and understand how well they work as a precursor to parameterized algorithms for solving classical problems like Vertex Cover and Feedback Vertex Set. We hope this study adds to the larger goal of bridging the gap between theoretical analysis of parameterized algorithms and algorithm engineering.

Contents

1 Introduction **1**

1.1 Graph Preliminaries 1

1.2 Vertex Cover and Feedback Vertex Set(FVS) 3

1.3 Organization of The Report 3

2 Parameterized Complexity **5**

2.1 Definitions 5

2.2 Algorithm Design Techniques 5

3 Vertex Cover and FVS **6**

4 Conclusion and Future Work **15**

References **16**

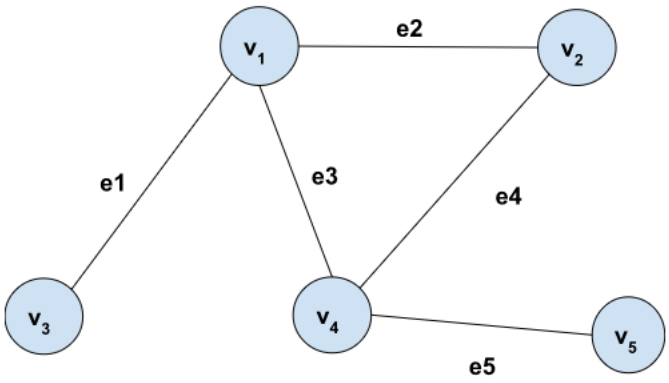
Chapter 1

Introduction

Graph is a type of data structure which is used in solving various complex problems. Real-world applications of graph theory include the internet, social media, web page searching, city planning, traffic control, transportation, and navigation, the travelling salesman problem, GSM mobile phone networks, map colouring, schedule scheduling, and more. In this chapter we look at basic structure and properties of graph.

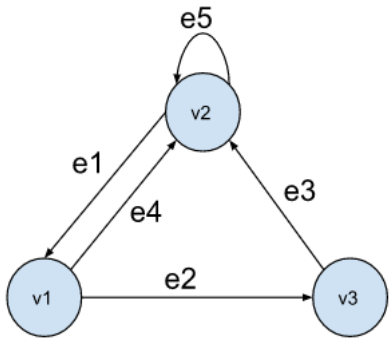
1.1 Graph Preliminaries

A graph G can be considered as a set of collections of V and E where V is a finite set of vertices and E is a set of edges between the vertices $E \subseteq \{(u, v)|u, vV\}$. The vertex set of a graph G is referred to as V (G), its edge set as E(G)
A graph G is denoted as G = (V,E)

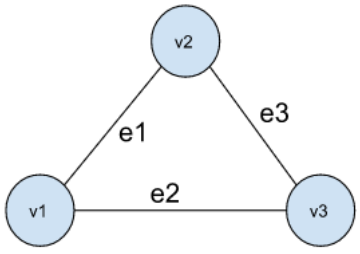


Types of graphs :

1)Directed graphs (digraph) : The edges are digraphs are directed edges i.e we can only traverse in the direction they are pointing.



2)Undirected graphs : The edges here are undirected so traversal in both directions is possible.



A simple graph is a graph in which the edges are undirected and there are no multiple edges between any two vertices. Also there are no loops present in simple graphs.

A weighted graph is a graph in which the edges have some weights to them.

Consider a graph G=(V,E) and let $v \in V$, the degree of v (represented as deg(v)) is the

number of non-self loop edges adjacent to the vertex v plus two times the number of self loops present at vertex v.

A path P in a graph is defined as a sequence of vertices $v_1, v_2, v_3, ...v_k$ such that $v_i v_j$ (here j = i+1) is an edge for each i=1,2,3,..k-1. The length of the path P is the number of edges in P.

A graph G is said to be connected if we can traverse from any vertex $u \in V$ to any vertex $v \in V$. If a graph is not connected then it is considered disconnected.

A graph G is said to be sparse if for every $u \in V$ we have degree of $u < n/2$. Where n is the number of edges close to the maximal number of edges.

A graph G is said to be dense if for every $u \in V$ we have degree of $u > n/2$. Where n is the number of edges close to the maximal number of edges.

1.2 Vertex Cover and Feedback Vertex Set(FVS)

Vertex Cover :

A vertex cover of a graph G is a collection of vertices that includes at least one end point of every edge in graph G.

Feedback Vertex Set:

The feedback vertex set of a graph is a set of vertices which when removed makes the graph cycleless (i.e cycles will be removed from the graph).

1.3 Organization of The Report

This chapter provides a background for the topics covered in this report. We provided a description of Graph preliminaries and vertex cover and Feedback Cover Vertex (FVS). The rest of the chapters are organised as follows: In Chapter II we provided review of parameterized complexities and a few Algorithm Design Techniques in detail. In Chapter III we discussed about the Vertex cover and Feedback vertex set and proved the same

in very detail. And finally in chapter IV, we concluded with some future works that are required to be done.

Chapter 2

Parameterized Complexity

2.1 Definitions

Generally many problems we want to solve are NP-hard, but we want to solve these problems. Over a few years a new paradigm has been introduced where the time complexity of an algorithm is measured not just in terms of input length but also a side parametre. The goal here is to find the parametrization of hard problems so that we can design algorithms running in polynomial time from input length to exponential (worse) in small parametre. Such algorithms are called parameterized algorithms.

2.2 Algorithm Design Techniques

Kernilization :

I is an instance of a decision problem with parameter P. A parameterized problem is an instance of a decision problem with a parametre P. Consider an example of vertex cover. The instance I will contain a graph G and integer K. Depending on what parameter we want to deal with, with vertex cover, P will be that parametre. We work with a parametre which is solution size K. So P will be K. Given an instance (I,P), a kernelization algorithm is a polynomial time algorithm that takes input (I,P) and returns output (I',P').

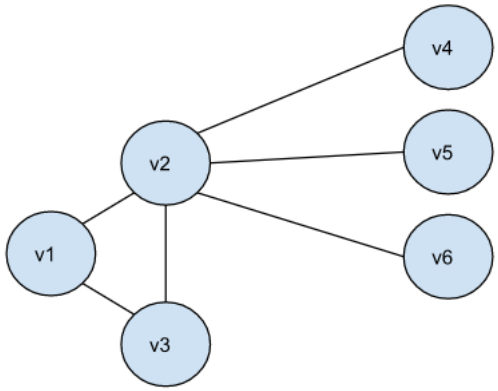
Chapter 3

Vertex Cover and FVS

- **Vertex Cover**

  A vertex cover S for a graph G = (V,E) is a subset of V(G) i.e $S \subseteq V(G)$ such that for $uv \in V(G)$ either $u \in S$ or $v \in S$.

  A **minimum vertex cover** is a vertex cover which contains least possible vertices.



  Here S ={v1,v2,v4} is a vertex cover but S = {v2,v3} is a minimum vertex cover. There are two types of problems we face in this topic. They are **decision** based and **optimization** based. In optimization problem we find the best possible solution for example minimum vertex cover comes in this category as we are trying to find the smallest possible set which is a vertex cover whereas in decision based

problem we are required to give an algorithm which when fed with input gives out answer as either 'yes' or 'no' for example given an instance of graph G does G have a vertex cover of at most size 'k'.

The decision variant problems of vertex cover are NP-complete i.e the chance of solving the problem in an efficient manner for a given arbitrary graph is unlikely. It is frequently used in the theory of computer complexity as a foundation for NP-hardness arguments.

The vertex cover problem is **fixed-parametre tractable** i.e an exhaustive search/brute force method can solve the problem in $2^k n^c$.

Here it is fixed parametre tractable for small values of k hence we can solve it in polynomial time. Here 'k' is referred as **klam value**(it is the value which bounds the parameter value for which the algorithm will work in practical limits).The algorithm for solving vertex cover that achieves the best asymptotic dependence on the parameter runs in time O(1.2378^k + (k.n)) and the klam value for it is approximately 190[1].

Algorithms :

- **Naive Approach :**

  Here in this approach we will consider all the subset of vertices of the graph. We iterate over all the subset of vertices one-by-one and check if it covers all the edges of the graph. After doing this we get the subsets which cover all the edges of the graph and from them we will take the minimum size subset as we require minimum size vertex cover. Finally the set with minimum size is printed. The run-time complexity for this approach is exponential (2^n) where n is the number of vertices of the graph because each vertex of the graph has 2 choices of either picking it or not picking it. We check for all the subsets and print the minimum size set that covers all the edges of the graph.

Here we first generate all 2n subsets where 'n' is number of vertices in the graph. Let the set which stores all subsets be 'S' and 'E' be the edge set and 'CheckVC' is a function which checks whether we get independent set after we remove a subset from E.

```
X = n #Variable to store the size of Vertex Cover
for i in S
    If ( CheckVC(E - S(i)) == True)
        X= min(X,sizeof(S(i))
    Else
        continue
```

Fig. 3.1 Brute Force Algorithm

At the end we get the minimum possible vertex cover.

– **Branching Algorithm :**

This algorithm is based on a simple fact that if we consider a vertex v of a graph G, the vertex must contain either v or all of the neighbors of v. The neighbors of v are represented by N(v). If the maximum degree of G is at most 1, then vertex cover can be solved in polynomial time.

Now we are given an instance (G,k), the first step is to find the vertex v(G) with maximum degree in G. Now, if the degree of v is 1, then it means that there are isolated vertices or an edge. This instance will have a trivial solution. Now, if the maximum degree of a vertex in G is 2, we recursively branch either by taking v in the vertex cover or N(v) in the vertex cover. Based on the branching, the instance would change to either (G,k-1) or $(G, k - |N(v)|)$. This algorithms run time is bounded by product of number of nodes in the search tree of the algorithm and time taken at each node.

While running the branching algorithm, each internal node in the search tree will have at least 2 child nodes.So, if there are l leaves in the tree then

number of nodes in the tree is at most 2l -1.

We can define the recursive relation as T(k)=T(k-1)+T(k-2) if $k >= 2$, otherwise T(k)=1. With the help of induction we can prove that T(k) is bounded by 1.6161^k.

The recursive relation in this case where we eliminate degree 2 and make sure the graph has at least 3 degree will be T(k)=T(k-1)+T(k-3) if $k >= 3$ or else T(k)=1. The run time for this algorithm will be bounded by 1.4656^k.

– **2-Approximation algorithm :**

Consider a graph G with E as the set of edges. In this algorithm we go through the edge set while it is not empty and we add edge (u,v) to our result and then we delete all the edges incident on either 'u' or 'v' from E and repeat the loop.

As the set of edges picked by the approximation algorithm is a matching as no 2 edges are touching each other so it is a maximal matching.

This algorithm guarantees us a vertex cover of size at max twice the size of minimal vertex cover.

Let us consider our vertex cover set be denoted as VC and E as edge set. Let $\{u, v\}$ denote and edge in E. Then our algorithm works as below.

```
VC ← ∅ #Initially our Vertex Cover set is empty
while E ≠ ∅  #Run the loop until the edge set becomes empty
        pick any {u, v} ∈ E
        VC ← VC ∪ {u, v} # adding u,v to VC
        delete all edges from u and v from E
return C
```

Fig. 3.2 2-approximation algorithm

Above is the algorithm for 2-approximation.

– **C(n,k) Algorithm:**

Here we are given a graph G which has 'n' number of vertices and we need find out if a vertex cover of size 'k' exists.

In this approach we select 'k' vertices in a given a 'n' number of vertices and check if the satisfies the condition of vertex cover i.e if we remove the selected 'k' vertices then we must get a graph where each vertex is isolated. If we get yes as our output we then stop our algorithm else we continue until we get a solution and if all possible options are explored and if we don't get a solution we can say that for the given graph there is no vertex cover of size k.

This algorithm is much faster compared to brute force as the runtime here reduces significantly as 2^n is much greater than C(n,k).

Here we first generate all C(n,k) combinations where 'n' is number of vertices in the graph and 'k' is the size of vertex cover. Let the set of combinations be 'S' and 'E' be the edge set and 'CheckVC' function checks whether we get an independent set after we remove a combination from E.

```
for i in S
    If ( CheckVC(E - S(i)) == True)
        return true;
    Else
        continue
```

Fig. 3.3 C(n,k) Algorithm

- **Vertex Cover LP Kernel:**

To implement a vertex cover LP-based kernel, we first need to understand what LP-based kernels are.

Given an instance of an optimization problem expressed as a linear program, an LP-

based kernel is a polynomial-time algorithm that generates a smaller instance that is equal to the original instance in terms of optimal solutions. Specifically, an LP-based kernel for vertex cover is an algorithm that, given an instance of the vertex cover problem expressed as a linear program, outputs an equivalent instance with a smaller number of variables and constraints.

There are many problems that can be expressed in the form of INTEGER LINEAR PROGRAMMING(ILP). In an instance of ILP we are given with a set of integer-valued variables, a set of constraints(Linear inequalities) and a linear cost function. We have to evaluate the variables in such a way that all the constraints are satisfied and the value of the cost function is maximized or minimized. In order to express a vertex cover instance as a ILP instance we required $|V(G)|$ number of variables where each variable x_v represents each vertex V(G). The value of x_v can be either 0 or 1 depending on whether it is taken into vertex cover or not. In order to make sure that every edge of the graph is covered we have the constraint $x_v + x_u \geq 1$ for every edge $uv \in E(G)$. By using all of these we obtain the following ILP:

minimize $\sum_{v \in V} x_v$

Constraints : $x_v + x_u \geq 1$ for every uv $\in E(G)$

$0 \leq x_v \leq 1$ for every v $\in V(G)$

$x_v \in Z$ for every $v \in V(G)$.

We cannot solve the ILP in polynomial time as it is as hard as the vertex cover.Hence to solve this problem we approach in a different way by relaxing the integer constraint of ILP. By doing this we can reduce the hardness of the problem and we obtain LINEAR PROGRAMMING(LP). The instances of LP look the same as ILP but here the variables can take any real values instead of only integers. We remove the constraint $x_v \in Z$ for every $v \in V(G)$.

We call this relaxation on graph G as LPVC(G). Taking fractional values corresponds

to taking some part of vertex v into the vertex cover. Though the LPVC(G) does not exactly express the vertex cover problem on graph G, its optimum solution is useful in knowing about the minimum vertex covers of graph G. Even in the case where there are many constraints, LP can be solved in Polynomial time. An optimum solution obtained by relaxation LPVC(G) will contain the variables corresponding to the V(G) whose values are in between 0 and 1.

We can divide these vertices of G into 3 sets based on the value of the variable as follows:

V_0 will contain all the variables whose value is less than $\frac{1}{2}$.

$V_{1/2}$ will contain all the variables whose value is equal to $\frac{1}{2}$.

V_1 will contain all the variables whose value is greater than $\frac{1}{2}$.

The Nemhauser-Trotter Theorem tells us that there exists a minimum vertex cover OPT of G such that $V_1 \subseteq OPT \subseteq V_1 \cup V_{1/2}$ i.e OPT is completely contained inside V_1 and $V_1 \cup V_{1/2}$ and does not contain any vertex in V_0.

Now, let us consider $OPT' \subseteq V(G)$ be the minimum vertex cover of G. We define S in the following way, $OPT = (OPT' \setminus V_0) \cup V_1$. i.e we deleted all the vertices from V_0 and added every vertex in V_1. V_0 is an independent set because the end points of edges from V_0 are only present in V_1 as only this obeys the constraint $x_v + x_u \geq 1$ for every $uv \in E(G)$. So, OPT is a vertex cover and it is of at least OPT'. Justifying OPT as the minimal vertex cover is sufficient.

Now, let us consider a set (X_v) where $V \in V(G)$ is an optimum solution to LPVC(G) for a vertex cover instance (G,k). We can infer that we are dealing with a no-instance if $\sum_{v \in V(G)} x_v > k$. Or else take the vertices from V_1 greedily into the vertex cover. In other words, eliminate every vertex from $V0 \cup V1$ and reduce k by $|V1|$. It is obvious that the best answer to LPVC(G) is of cost at most k if (G, k) is a yes-instance. If we come to the conclusion that (G, k) is a no-instance, this shows that the step was done correctly.

Let us consider $G' = G - (V_0 \cup V_1) = G[V_{1/2}]$ and $k - |V_1| = k'$. If and only if (G', k') is a yes-instance of Vertex Cover, then (G, k) is, according to our assertion. By using the Nemhauser-Trotter Theorem we can say that G has a vertex cover S of size at most k such that $V_1 \subseteq S \subseteq V_1 \cup V_{1/2}$. $S' = S \cap V_{1/2}$ is a vertex cover in G' and the size of it is at most k'.

Now let us go in the opposite way i.e let S' be a vertex cover in G'. Every edge with an endpoint from V_0 should have an endpoint in V_1 for every solution of LPVC(G). As a result, $S = S' \cup V1$ is a vertex cover in G, and its size is at most $k' + |V_1| = k$. The above mentioned reduction will help us in leading to a kernel for vertex cover. The instance (G,k) is a vertex cover instance. After applying the above mentioned reduction rule on (G,k) we obtain a solution set (X_v)$where$V $\in V(G)$ by determining whether we are encountering a no-instance or acquiring an instance (G', k').

We can also observe the following:

$$|V(G')| = |V_{1/2}| = \sum_{v \in V_{1/2}} 2x_v \leq 2 \sum_{v \in V(G)} x_v \leq 2k$$

- **Feedback Vertex Set(FVS)**

  The feedback vertex set of a graph is a set of vertices which when removed makes the graph cycle-less (i.e cycles will be removed from the graph).

  Given a graph G and its set of vertices is represented by V(G) and $X \subseteq V(G)$. If G-X is a acyclic graph, then X is feedback vertex set of G. In the FVS problem, we have an undirected graph G and a non-negative integer k, and the goal is to see if G has an FVS of size ¡=k. G may contain multiple edges and loops. Double edge and loop are considered as a cycle. Any vertex with a loop will be contained in any solution set X. The feedback-vertex set problem can be solved using branching algorithm in time $k^{O(k)}\ n^{O(1)}$. Some reduction rules are applied to clean the graph.[2]

  Reduction 1: If there is a vertex v which has loop associated with it,then the

vertex v can be deleted from the graph and we can reduce the value of k by 1.

Reduction 2: Reduce the multiplicity of an edge by 2 if its multiplicity is greater than 2.

Reduction 3: If there are nodes with degree at most 1, they do not participate in any cycle of G and can be removed.

Reduction 4: Delete the vertices in the graph whose degree is 2 and connect the neighbours of v with a new edge.

If the vertex v that we want to reduce using the rule 4 has a loop associated with it, then reduction rule 1 should be used instead of reduction rule 4 on the vertex v. After using the above reduction rules, the resultant graph G does not have loops and the minimum degree of vertices in G is at least 3.

Reduction 5: End the algorithm if $k < 0$ and conclude that (G,k) is a no-instance. Let $(v_1, v_2, v_3 \ldots v_n)$ be the vertices of a graph G V(G) in descending order of vertex degree and let $V_{3k} = \{v_1, \ldots., v_{3k}\}$.

Let X be the feedback vertex set of G, then for all X of G, we have

$\sum_{v \in X} (d(v) - 1) >= |E(G)| - |V(G) + 1$

An algorithm with a running time of $(3k)^k n^{O(1)}$ also exits for feedback vertex set.

Chapter 4

Conclusion and Future Work

As mentioned in chapter 3 our next work is to understand and implement the algorithm listed. Then we plan to run the algorithms on standard benchmarks instances and document the results obtained. After that we plan to use kernelization and then go through benchmarks again and document the effects of kernelization on the runtime We hope this study not only helps us to understand how well theoretical algorithms work in practice but also inspires new theoretical developments for practical purposes.

References

[1] E. W. Weisstein, "Vertex cover wikiwand," *Wikiwand.com/en/vertex$_c$over*.

[2] Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh, "Parametrized algorithms," *Wikiwand.com/en/vertex$_c$over*, *vol*. $1, no.\ 3, pp.57 - -59, 2015$.