Experimental Analysis of Fixed Parameter Algorithms for Odd Cycle Transversals and Feedback Vertex Sets

S. Shivaprasad¹, Dr. M Sadanandam², Anand Agrawal³
¹,³Assistant Professor, Vignan’s Univeristy, INDIA
²Assistant Professor, Kakatiya University, INDIA

ABSTRACT

As implementing approximation algorithms for NP-hard problems seems to be impractical, a fixed-parameter algorithm is one that provides an optimal solution to a discrete combinatorial problem. The fundamental idea is to restrict the corresponding, seemingly unavoidable, combinatorial explosion that causes the exponential growth in the running time of certain problem-specific parameters. An odd cycle transversal (OCT) is a subset of the vertices of a graph G that hits all the odd cycles in G. Clearly the deletion of such a vertex set leaves a bipartite graph. Thus the problem of finding an odd cycle transversal of minimum cardinality is just the classical graph bipartization problem. A feedback vertex set F (briefly, FVS) for G is a set of vertices in G such that every directed cycle in G contains at least one vertex in F, or equivalently, that the removal of F from the graph G leaves a directed acyclic graph (i.e., a DAG). Odd Cycle Transversal and Feedback Vertex Sets are the NP-Hard problems which can be practically implemented by fixed parameter algorithms. This report is an study of algorithms for OCT and FVS. The study shows the influence of various factors, such as number of nodes, optimum solution size, edge density, and parameter size, on the performance of the FPT algorithm.

Keywords---- Graph, Odd Cycle Transversals, Feedback vertex Sets, Bipartization.

I. INTRODUCTION

A. Parameterised Algorithms

There has been a long history of exponential-time algorithms for finding optimal solutions to NP-hard problems. Exponential running time at first glance seems to be impractical. This conception has been challenged by the view of parameterized complexity. Parameterized Algorithms and Complexity is a natural way to cope with problems that are considered intractable according to the classical P versus NP dichotomy. In practice, large instances of NP-complete problems are solved exactly every day. The reason for why this is possible is that problem instances that occur naturally often have a hidden structure. The basic idea of Parameterized Algorithms and Complexity is to extract and harness the power of this structure. In Parameterized Algorithms and Complexity every problem instance comes with a relevant secondary measurement k, called a parameter. The parameter measures how difficult the instance is, the larger the parameter, the harder the instance. The hope is that at least for small values of the parameter, the problem instances can be solved efficiently. This naturally leads to the definition of fixed parameter tractability. We say that a problem is fixed parameter tractable (FPT) if problem instances of size n can be solved in f (k)nO(1) time for some function f independent of n.

Over the last two decades, Parameterized Algorithms and Complexity has established itself as an important subfield of Theoretical Computer Science. Parameterized Algorithms has its roots in the Graph Minors project of Robertson and Seymour [1], but in fact, such algorithms have been made since the early seventies [8, 9]. In the late eighties several FPT algorithms were discovered [10, 11] but despite considerable efforts no fixed parameter tractable algorithm was found for problems like Independent Set and Dominating Set. In the early nineties, Downey and Fellows [13, 12] developed a framework for showing infeasibility of FPT algorithms for parameterized problems under certain complexity theoretic assumptions, and showed that the existence of FPT algorithms for Independent Set and Dominating Set is unlikely. The possibility of showing lower as well as upper bounds sparked interest in the field, and
considerable progress has been made in developing methods to show tractability and intractability of parameterized problems. Notable methods that have been developed include Bounded Search Trees, Kernelization, Color Coding and Iterative Compression[14].

B. Basic Definitions:

Basic Notions : We assume that the reader is familiar with basic notions like sets, functions, polynomials, relations, integers etc.

Problems: A problem or language is a set L of strings over a finite alphabet Σ, that is L ⊆ (Σ)*. A string s ∈ L is a yes instance of L and a string sinLis a no instance of L. A parameterized problem is a subset I of (Σ)* x N, that is, every instance of Π comes with a natural number called the parameter. In an optimization problem we are given as input an instance s, defining a set F(s) of feasible solutions for a problem specific function F. The objective is to find an element x ∈ F(s) that maximizes or minimizes a certain problem specific objective function g(x). If the objective is to minimize (maximize)g(x) the problem is referred to as a minimization (maximization) problem. The minimum (maximum) value of the objective function over all the feasible solutions for an instance s is called the optimal value of the objective function.

Function Growth : We employ big-Oh notation which suppresses constant factors and lower order terms. That is, for two functions f : N → N and g : N → N we say f(n) = O(g(n)) if there are constants c0, c1 > 0 such that for all x < c0, c.f (x) < c1 g(x). We say that a function f is singly exponential if f = O(2^c n) for some constant c. If f = O(2^c n) for some constant c we say f is doubly exponential. We will sometimes misuse notation and say that f(n) is at least O(g(n)), meaning that g(n) = O(f(n)).

NP-hard problem : NP-hard (non-deterministic polynomial-time hard), in computational complexity theory, is a class of problems that are, informally, "at least as hard as the hardest problems in NP". A problem H is NP-hard if and only if there is an NP-complete problem L that is polynomial time Turing-reducible to H (i.e., LTH). In other words, L can be solved in polynomial time by an oracle machine with an oracle for H. Informally, we can think of an algorithm that can call such an oracle machine as a subroutine for solving H, and solves L in polynomial time, if the subroutine call takes only one step to compute. NP-hard problems may be of any type: decision problems, search problems, or optimization problems.

Algorithms : An algorithm for the problem L is an algorithm that for a string s determines whether s ∈ L. The running time of the algorithm is measured in the number of steps the algorithm performs. We will assume a single processor, random-access machine as the underlying machine model throughout this thesis. In the random-access machine any simple operation (arithmetic, if-statements, memory-access etc.) takes unit length of time, and the word size is sufficiently large to hold numbers that are singly exponential in the size of the input.

Graph : A graph G, can be defined as a tuple, G = (V, E), where V is a finite set and E is a collection of unordered pairs of distinct elements of V. The elements of V are called vertices of G and the elements of E are called edges of G. Let S denote the cardinality of set S. We denote |V| by n and |E| by m. We consider all graphs, we deal with, to be finite, undirected and unweighted, without multiple edges.

Odd Cycles : In graph theory, the term cycle may refer to a closed path. If repeated vertices are allowed, it is more often called a closed walk. If the path is a simple path with no repeated vertices or edges other than the starting and ending vertices, it may also be called a simple cycle, circuit, circle, or polygon A cycle graph or circular graph is a graph that consists of a single cycle, or in other words, some number of vertices connected in a closed chain. A cycle with an odd number of vertices is called an odd cycle.

Graph bipartization : Given an undirected graph G = (V, E) and a nonnegative integer k. Does G have an odd cycle cover C of size at most k, that is, is there a subset C ⊆ V of vertices with |C| ≤ k such that each odd cycle in G contains at least one vertex from C? Note that the removal of all vertices in C from G results in a bipartite graph.

C. Odd Cycle Transversal

An odd cycle transversal (or cover) is a subset of the vertices of a graph G that hits all the odd cycles in G. Clearly the deletion of such a vertex set leaves a bipartite graph. Thus the problem of finding an odd cycle transversal of minimum cardinality is just the classical graph bipartization problem. The objective is to find a set S of at most k vertices whose deletion makes the graph bipartite, and a set S such that G\S is bipartite is called an odd cycle transversal of G.

Input: A graph G and an integer k.
Output: A vertex set X ⊆ V(G) with |X| ≤ k such that G-X is bipartite.

D. Feed Back Vertex Sets

Feed Back Set problems deal with destroying cycles in graphs using a minimum number of vertex deletions or edge deletions.

Feedback Arc Set (FAS) asks for a minimum number of arcs to be deleted in order to obtain a cycle-free directed graph. Feedback Vertex Set (FVS) asks for a minimum number of vertex deletions. A tournament is a directed graph where between any two distinct vertices there is exactly one arc.
1. Register allocation for processors: Recent application for Graph Bipartization is in register allocation for processors that, to save wiring, have their register set divided into two banks and require the two operands of an instruction to reside in different banks [5]. Conflicts are modeled by a graph where vertices correspond to operands and edges connect operands that occur together in an instruction. A minimum graph bipartization set then yields the minimum size set of operands that have to be copied into both banks to be able to execute the code.

2. SNP haplotyping: Another application originates from computational biology. To determine gene sequences, for technical reasons the DNA is first broken into small fragments (shotgun sequencing), from which the original sequence is reconstructed by computer[4] This is complicated by the fact that each gene occurs twice in the human genome. The two copies are mostly identical, but differ at certain sites (so-called SNPs). Given a set of gene fragments, the problem of assigning the fragments to one of these two copies in a consistent manner while dismissing the least number of SNPs as erroneous is called the Minimum Site Removal problem. The Minimum Site Removal problem can be solved using Graph Bipartization algorithms.

3. Deadlock recovery on Operating Systems: An important application of the FVS problem is deadlock recovery in operating systems, in which a deadlock is presented by a cycle in a system resource-allocation graph G.[7] In order to recover from deadlocks, we need to abort a set of processes in the system, i.e., to remove a set of vertices in the graph G, so that all cycles in G are broken. Equivalently, we need to find an FVS in G. The problem also has a version on weighted graphs, where the weight of a vertex can be interpreted as the cost of aborting the corresponding process. In this case, we are looking for an FVS in G whose weight is minimized.

4. Deadlock Detection: Deadlock recovery in concurrent programs has always been considered an important application of DFVS [1]. Indeed, deadlock recovery is a very important topic in our modern multi-core/many-core computing era. Solving the concurrency problem has recently seen tremendous research interest in operating systems, programming lan-guages and computer architecture communities.

For simplicity of analysis we mainly focus on mutex locks in the POSIX Thread library (Pthread). We will argue that, with the restrictions of the programming model, DFVS is not more helpful than cycle detection for deadlock recovery. This proposition is further confirmed by a report on a deadlock immunity system in a recent systems conference.

In a concurrent program, the nodes of a Resource Allocation Graphs (RAG) are resources and threads. Resources are simply mutex locks. There are three types of edges in a RAG: request, grant and own. The request and grant edges are edges from thread to lock, while the own edges go from lock to thread. Request edge means a thread is requesting a lock, and grant edge means the thread library allows the thread to wait on the lock (note that the thread may yield its execution to another one before being allowed to wait on the resource). The own edge from lock to thread means the thread currently owns this resource exclusively. A deadlock appears as a cycle in the RAG. In such a cycle, all edges are exclusively the grant and own edges. One lock can be owned by only one thread, even for recursive locks, though certain threads could acquire the lock multiple times. One thread, at one time, can be granted to wait on only one lock because threads are executed sequentially, and we cannot wait on two resources simultaneously. Thus, we cannot have overlapping cycles in a RAG, and therefore simple cycle detection suffices to resolve the deadlocks.

5. Constraint satisfaction and Bayesian inference in artificial intelligence and graph theory.

II. LITERATURE SURVEY

A. Overview

In this a detailed survey of all the algorithms proposed by different authors to solve Odd Cycle Transversals and Feedback Vertex Sets are listed in the first section. It is followed by a table which is like a comparison between the algorithms based on different parameters like time complexity and number of edges.

B. Approaches to the problem

1. Brute Force Approach: We can, trivially, determine whether a graph has an odd cycle transversal containing k vertices via brute force enumeration in O(nmk).

2. Bipartization in polynomial Solvable time: A set W of vertices is an odd cycle transversal of G precisely if every face of (G - W) contains an even number of odd faces of G. We find a Planar Graph Bipartition problem in linear time by using the term 'parity of the face' of the planar graph. We consider an embedding of the planar graph G. The parity of a face of G is defined as the parity the edge set of its boundary, counting bridges twice. The crucial observation is that the parity of a cycle in G is equal mod 2 to the sum of the parities of the faces within it. In particular, it follows from the crucial observation that G is bipartite if and only if all its faces are even. When a vertex v is deleted from G, all the faces incident to v are merged together in a new face F. The other faces are unchanged. We denote the new face by a capital letter to stress the fact that it determines a set of
faces of $G$, namely, the faces of $G$ included in it. Note that the parity of the new face $F$ equals the sum mod 2 of the parities of the faces of $G$ it contains.

Let now $W$ denote any set of vertices in $G$. By deleting from $G$ the vertices in $W$ one after the other in some order, we see that each face of $G-W$ corresponds to a set of face of $G$. This set is a singleton if the corresponding face is a face of $G$ that survived in $G-W$. Furthermore, a face of $G-W$ is odd precisely if it contains an odd number of odd faces of $G$. Because a planar graph is bipartite if and only if all its faces are even.

3. Fixed Parameter Tractability: As we can, trivially, determine whether or not a given graph has an odd cycle transversal containing at most $k$ vertices via brute force enumeration in $O(mn)$ time. Although this is polynomial time for each fixed $k$, it is practically too slow for large inputs, even if $k$ is relatively small. Therefore, the standard goal of parameterized analysis is to take the parameter out of the exponent in the running time. A problem is called fixed-parameter tractable (FPT) if it can be solved in time $O(f(k)n^p)$, where $p$ is a constant not depending on $k$, and $f$ is an arbitrary function. An algorithm with such a running time is also called FPT.

Iterative Compression: The idea is to reduce the problem in question to a modified version, where we are also given as input a solution that is almost good enough, but not quite. For the case of Odd Cycle Transversal, we are given an odd cycle transversal $S^0$ of $G$ of size $k + 1$. We call this problem the compression version of Odd Cycle Transversal. The crux of the Iterative Compression method is that the compression version of a problem is easier to solve than the original one. Suppose we could solve the compression version of the problem in $O(f(k)n^c)$ time. We show how to solve the original problem in $O(f(k)n^{c+1})$ time. Order the vertices of $V(G)$ into $v_1 v_2 \ldots v_n$ and define $V_i = v_1 \ldots v_i$ for every $i$. Notice that if $G$ has an odd cycle transversal $S$ of size $k$ then $S \cap V_i$ is an odd cycle transversal of $G[V_i]$ for every $i$. Furthermore, if $S$ is an odd cycle transversal of $G[V_i]$ then $S \cap V_i+1$ is an odd cycle transversal of $G[V_i+1]$. Finally, $V_k$ is an odd cycle transversal of size $k$ of $G[V_k]$. These three facts together with the $f(k)n^c$ algorithm for the compression version of OCT give a $f(k)n^{c+1}$ time algorithm for OCT as follows. Call the algorithm for the compression version with input $(G[V_k+1], V_k+1, k)$. The algorithm will either report that $(G[V_k+1], k)$ has no odd cycle transversal of size $k$ or return such an odd cycle transversal, call it $S_{k+1}$. In the first case $G$ has no $k$-sized odd cycle transversal. In the second, call the algorithm for the compression version with input $(G[V_k+2], S_{k+1} \cap V_k+2, k)$. Again we either receive a no answer or a $k$-sized odd cycle transversal $S_{k+2}$ of $G[V_k+2]$ and again, if the answer is negative then $G$ has no $k$-sized odd cycle transversal. Otherwise we call the compression algorithm with input $(G, S_{k+2} \cap V_k+3, k)$ and keep going on in a similar manner.

C. Existing Algorithms

The following is a table that shows the existing algorithms for Graph Bipartization using Odd Cycle Transversals. Of these algorithms, the algorithm given by Huffner [6] algorithm which is an improved version of Bruce, Reed’s [2] algorithm is considered in the thesis.

<table>
<thead>
<tr>
<th>Author</th>
<th>Year</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bruce, Reed, Katoji, Sazh, Adachi, Yama</td>
<td>2006</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Saurav, V Naras, Ravi, Ritter</td>
<td>2008</td>
<td>$O(n^2 \log n)$</td>
</tr>
<tr>
<td>Reckoni, Kasanevski, Bruce Reed</td>
<td>2009</td>
<td>$O(nm + n)$</td>
</tr>
<tr>
<td>Dado, Lodhakarn, Iwamoto, Saito, Yama</td>
<td>2009</td>
<td>$O(n^2 \log n)$</td>
</tr>
<tr>
<td>Fall, Hutter</td>
<td>2008</td>
<td>$O(n^3)$</td>
</tr>
</tbody>
</table>

Table 2.1: List of existing algorithms for OCT comparison.

D. Related Work to FVS

It is reasonable to study feedback set problems from a parameterized point of view [15, 16, 17] (see also Gutin and Yeo [18] for a survey on parameterized problems on directed graphs). For instance, in undirected graphs, using iterative compression [19], it has been shown that a feedback vertex set of size at most $k$ can be found in $37.7^k nO(1)$ time [20] and $10.57^k nO(1)$ time [36], where $n$ is the number of graph vertices. The running time has now been lowered to $5^k n^{2}$ [22]. The question whether Feedback Vertex Set on general directed graphs is fixed parameter tractable had been famously open for a long time and has only recently been resolved positively, also using iterative compression [33]; however, the given algorithm running in $4^k k^n nO(1)$ time incurs a much worse combinatorial explosion with respect to the parameter $k$ than those algorithms specialized to tournaments. Restricting the consideration to the
class of tournaments, Raman and Saurabh [24] have given the first positive result by giving fixed-parameter algorithms for weighted FAST and weighted FAST running in $2.415^knO(1)$ time. For the unweighted case of FAST, the previously fastest algorithm is obtained by a reduction to 3-Hitting Set and runs in $2.076^knO(1)$ time [25]. In a recent manuscript, Alon, Lokshtanov, and Saurabh [26] gave for FAST one of the rare examples of a sub-exponential time algorithm with a running time of $2^{O(k + \log^2 k)} + nO(1)$ . An algorithm for FVS BT with a running time of $O(3.116^k n^4)$ can also be derived using a 4-Hitting Set algorithm by Fernau [27]; with the 4-Hitting Set algorithm, we get a running time of $O(3.076^k + n^4)$. Recently, an algorithm for FVS BT running in $O(3kn^2 + n)$ time was given by Sasatte [28].

E. Major Approaches

1. **Iterative Compression** : As already explained the key idea of Iterative Compression is to construct size-k solutions from already known size-(k + 1) solutions.

2. **Kernelization** : A kernelization is an efficient algorithm that preprocesses instances of decision problems by mapping them to equivalent instances with a guaranteed upper bound on the size of the output, called the kernel of the instance. Kernelization is often achieved by applying a set of reduction rules that cut away parts of the instance that are easy to handle. A kernelization replaces, in polynomial time, an instance by a decision equivalent instance (the kernel) whose size can be bounded by a function of the parameter $k$, that is, it will not depend on the original problem size $n$ anymore. Using kernels for d-Hitting Set [1], one can derive a kernel of $O(k^2)$ vertices and $O(k^3)$ edges for FAST, and a kernel of $O(k^2)$ vertices and $O(k^3)$ edges for FVS BT. A kernel for FAST is also easy to achieve: If an arc occurs in more than $k$ triangles, it needs to be deleted. After doing this exhaustively, at most $O(k^2)$ vertices can be left, or the instance is unsolvable.

### F. Existing algorithms

The following is a table that shows the existing algorithms for weighted FVS problem.

<table>
<thead>
<tr>
<th>Author</th>
<th>Year</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>H.Bodlaender</td>
<td>1994</td>
<td>$O(17k(2^n))$</td>
</tr>
<tr>
<td>V.Kann, Saurabh</td>
<td>2002</td>
<td>$O(n17k(2^n)^2)$</td>
</tr>
<tr>
<td>E. Kufner, R. Porti 1994</td>
<td>2004</td>
<td>$O(2^n + \text{poly}(n))$</td>
</tr>
<tr>
<td>V.Kann, Saurabh, Scholten</td>
<td>2004</td>
<td>$O(2^n + \text{poly}(n))$</td>
</tr>
<tr>
<td>X.Zhou, R. Porti 1994</td>
<td>2004</td>
<td>$O(2^n + \text{poly}(n))$</td>
</tr>
<tr>
<td>X.Lee, H. V. Frieze</td>
<td>2006</td>
<td>$O(2^n + \text{poly}(n))$</td>
</tr>
<tr>
<td>Michael Dem, Hang Guo, Falk Hufner, BartS</td>
<td>2009</td>
<td>$O(17k(2^n))$</td>
</tr>
</tbody>
</table>

Table 2.2: List of existing algorithms for FVS.

### III. ALGORITHMS ON FIXED PARAMETER TRACTIBILITY

#### A. Motivation for the Fixed Parameter Tractibility

It is unlikely to implement NP-hard problems in efficient time. For a simple example on a 1000 node graph, an $O(2^n)$ time algorithm takes $2^{1000}$ time which is highly impractical. So, a fixed-parameter algorithm is one that provides an optimal solution to such a discrete combinatorial problem. The fundamental idea is to restrict the corresponding, seemingly unavoidable, combinatorial explosion that causes the exponential growth in the running time of certain problem-specific parameters.

As another example of parameterization, consider the problem of placing as few queens as possible to attack all the squares on a chessboard. There is a way to place only five (which is optimal) queens on an $8 \times 8$ chessboard to do this. Here, a natural parameter is the size $k$ of the solution set we search for, that is, the set of queues to be placed. Hence for $8 \times 8$ chessboards $k = 5$. What about general $n \times n$ chessboards? Can we find a minimum solution efficiently? With the theory of NP-completeness at hand, we can prove meaningful statements about the computational complexity of problems. But what happens after we have succeeded in proving that a problem is NP-hard (that is, intractable), but which nevertheless must be solved in practice? In other words, how do we cope with computational intractability?

The approach followed is based on worst-case analysis of deterministic, exact algorithms to solve hard problems. Each approach has advantages and disadvantages. With regard to exact fixed-parameter algorithms, the advantages are guaranteed optimality of the solution and provable upper bounds on the computational complexity.

The disadvantage is that we have to take into account exponential running time factors. Clearly, exponential growth quickly becomes prohibitive when running algorithms in practice. Fixed-parameter algorithmics provides guidance on the feasibility of the exact algorithm approach for hard problems by means of a refined, two-dimensional complexity analysis.

#### B. Algorithm for Odd Cycle Transversal given by Falk Hufner [6]

Hufner’s algorithm is based on the “iterative compression method” introduced by Smith, Reed, and Vetta. The paper improves the earlier algorithm by using the Edmond Karp’s flow problem to the odd cycle transversal problem.

The algorithm is explained as follows, proofs are to be referred to the cited reference.
C. Top-Down Presentation of the Reed-Smith-Vetta Algorithm:

The global structure is illustrated by the function Odd-Cycle-Cover. It takes as input an arbitrary graph and returns a minimum odd cycle cover.

Algorithm:

Odd-Cycle-Cover(G = (V, E)) :
1: V ← φ
2: C ← φ
3: for each v in V 
4: do V ← V ∪ v
5: C ← Compress-OCC (G[V], C ∪ v)
6: return C

The routine Compress-OCC takes a graph G and an odd cycle cover C for G, and returns a smaller odd cycle cover for G if there is one; otherwise, it returns C unchanged. Therefore, it is a loop invariant that C is a minimum odd cycle cover for G[V], and since eventually V = V, we obtain an optimal solution for G. It remains to implement Compress-OCC. The idea is to use an auxiliary graph H(G, C) constructed from G = (V, E) as follows:

Remove the vertices in C from G and determine the sides of the remaining bi-partite graph. For each c ∈ C, add a vertex c1 to one side and another vertex c2 to the other side. For each edge v, c ∈ E with v ∈ C and c ∈ C, connect v to that vertex from c1 and c2 that is on the other side. For each edge c, d ∈ E with both c, d ∈ C, arbitrarily connect either c1 and d2 or c2 and d1.

Figure 3.1: Graph for algorithm for various parameters.

The crucial property of the resulting graph H is that every odd cycle in G that contains a vertex c ∈ C implies a path (c1, . . . , c2 ) in H. This means that all odd cycles in G can be found as such paths in H, since the vertices in C touch all odd cycles. For example, the triangle d, c, h in G can be found as path (c1 , h, d, c2 ) in H(G, C). Therefore, if we could find a set C of vertices whose removal disconnects for each c ∈ C the two vertices c1 and c2 in H, then C is an odd cycle cover for G.

Compress OCC(G,C):
1: for each Y ⊆ C 
2: do H ← Aux-Graph (G \ (C \ Y), Y) 
3: for each valid partition (Y1, Y2) of Y 
4: if ∃ vertex cut D in H between Y1 and Y2 with |D| ≤ |Y| 
5: then return(C \ Y) ∪ D
6: return C

We examine every subset Y of the known odd cycle cover C. For each Y, we look for smaller odd cycle covers for G that can be constructed by replacing the vertices of Y in C by fewer new vertices from V \ C (clearly, for any smaller odd cycle cover, such a Y must exist). Since we thereby decided to retain the vertices in C[Y] in our odd cycle cover, we examine the graph G = G \ (C \ Y). If we now find an odd cycle cover D for G with D ⊆ Y we are done, since then (C \ Y) ∪ D is an odd cycle cover smaller than C for G.

D. Running times

Reed [2] state the run time of their algorithm as O(4^k kmn); a slightly more careful analysis reveals it as O(3^k kmn). For this, note that in effect the two loops in line 1 and 3 of Compress-OCC iterate over all possible assignments of each c ∈ C to 3 roles: either c ∈ C \ Y , or c ∈ Y1 , or c ∈ Y2 . Therefore, we solve 3k flow problems, and since we can solve one flow problem in O(km) time by the Edmonds-Karp algorithm, the run time for one invocation of Compress-OCC is O(3^k kmn). As Odd-Cycle-Cover calls Compress-OCC n times, we arrive at an overall run time of O(3^k kmn).

E. FPT algorithm given for Feedback Vertex Sets

As per Huffner’s paper, the problem Feedback Vertex Set (Feedback Arc Set) in tournaments, FVST (FAST) for short, is defined as follows:

Input: A tournament T and a nonnegative integer k.

Task: Find a set F of at most k vertices (arcs) whose deletion results in an acyclic digraph.

The set F is called a feedback vertex set (feedback arc set). When the input digraph is restricted to bipartite tournaments instead of tournaments, we call the problem Feedback Vertex Set (Feedback Arc Set) in bipartite tournaments, FVSBT (FASBT) for short.

The Feedback Vertex Set (FVS) problem is a special case of the Hitting Set problem. Formally, the optimization version of the Hitting Set problem presents a set T with elements as subsets of a universal set X; the objective is to find a subset S ∈
X of elements of minimum cardinality so that every subset in T contains at least one element from S. One can find an easy factor d-approximation algorithm for the hitting set problem if the size of each subset in T is restricted to be at most d. The number of subsets to consider in T by an algorithm is potentially as high as \( o(2^{|X|}) \). I would like to find an algorithm that can judiciously pick a polynomial-sized choice of the subsets T to hit so that, once a hitting set S is picked for T, with high probability S is also the hitting set for T. While this is unlikely to work for all inputs (due to the NP-hardness of the problem) FBS is also solved by using Iterative Compression. The pseudo code is as follows

**Algorithm.** 1 Pseudo-code for iterative compression, using the compression routine Compress. The function call `Compress(G[V], X)` returns a hitting set for G[V] that is smaller than the hitting set X, if possible.

**ITERATIVE COMPARISON G(V,E):**

1. \( V \leftarrow \phi \)
2. \( X \leftarrow \phi \)
3. for each \( v \in V \) :
4. \( V \leftarrow V \cup v \)
5. \( X \leftarrow X \cup v \)
6. \( X \leftarrow \text{Compress}(G[V], X) \)
7. return X

**Iterative compression for hitting set:** Huffner solves the Hitting Set, which generalizes FVST and FVSBT, by using iterative compression. As introductory example, we use 3-Hitting Set. To emphasize the similarity to the graph problems, we formulate it as a hypergraph modification problem. 3-Hitting Set.

**Instance:** A hypergraph \( G = (V, E) \) with \(|e| = 3\) for all \( e \in E \) and an in-teger \( k \).

**Question:** Is there a hitting set \( X \subseteq V \) with \(|X| = k\), that is, a set of vertices whose deletion removes all hyperedges? Here, deleting a vertex implies also removing all hyperedges that contain this vertex. 3-Hitting Set is NP-complete. There is a simple 3-approximation for the minimization version of the problem (repeatedly take all three vertices of a hyperedge); it has been conjectured that this approximation factor cannot be improved [37]. Note that the variant 2-Hitting Set is equivalent to the NP-complete Vertex Cover problem. 3-Hitting Set can be solved in \( O(3^k) \) time by a simple search tree algorithm: choose any hyperedge \( v_1, v_2, v_3 \in E \) and branch into the three cases \( v_1 \in X, v_2 \in X \), and \( v_3 \in X \). By case distinction and careful analysis, this has been improved in a series of results to \( O(2.27^k + m) \) [41], then \( O(2.179^k + m) \) [21,22], and finally \( O(2.076^k + m) \). A kernel of size \( O(k^3) \) is known, which has recently been improved to \( O(k^2) \) vertices and \( O(k^3) \) edges [1].

The central idea is to use a compression routine, that is, an algorithm that, given a problem instance and a solution, either calculates a smaller solution or proves that the given solution is of minimum size. The most obvious way to employ a compression routine is to start with an approximate solution and then use the compression routine until no further compression is possible. However, since the running time of the compression routine depends exponentially on the size of the solution to compress, it is faster to build up the graph vertex-by-vertex while always keeping a minimal solution. This is illustrated in the pseudo-code below.

**Compress(G,V,X):**

1. for each \( S \subseteq X \):
2. \( D \leftarrow X \setminus S \)
3. if \( G[S] \) is a hyperedge-free graph:
4. \( G' \leftarrow G[V \setminus D] \)
5. \( S' \leftarrow \text{CompressDisjoint}(G', S) \)
6. if \(|S'| < |S|\):
7. return \((X \setminus S) \cup S'\)
8. return X

Figure 3.2: Pseudo-code for Compress. The function call `CompressDisjoint(G, S)` returns a hitting set for G that is smaller than the hitting set S, if possible, and disjoint from S.

We start with \( V = \phi \) and \( X = \phi \); clearly, X is a minimum hitting set for G[V]. In lines 4 and 5, we add one vertex \( v \in V \) from V to both V and X. Then X is still a hitting set for G[V], although possibly not a minimum one. We can, however, obtain a minimum one by applying our compression routine. Here, the compression routine Compress takes a hypergraph G and a hitting set X for G, and returns a smaller hitting set for G if there is one; otherwise, it returns X unchanged. Therefore, it is a loop invariant that X is a minimum-size hitting set for G[V]. Since eventually \( V = V \), we obtain an optimal solution for G once the algorithm returns X. Note that we defined a compression routine as a function that returns a smaller solution, but not necessarily a minimum one. This suffices here, because the hitting set X to be compressed can be larger by at most one than an optimal hitting set X for G[V]; this is because X is also a hitting set for G[V], and cannot be smaller than the minimum hitting set X. It remains to describe the compression routine. The basic idea, which is shared with most other known iterative compression algorithms [30], is to reduce the compression problem to a disjoint compression problem:
Iterative compression for feedback vertex set in tournaments

**FVST Disjoint Compression**

**Instance:** A tournament $T = (V, A)$ and a subset $S \subseteq V$ such that $T \setminus S$ and $T \setminus V \setminus S$ are acyclic.

**Task:** Find a set $S \subseteq V \setminus S$ with $|S| \leq |S|$ such that $T \setminus (V \setminus S)$ is acyclic.

Up to this point, the algorithm is analogous to the iterative compression algorithms for general directed Feedback Vertex Set [33] and undirected Feedback Vertex Set [36, 35]. The core part of the compression routine, however, is completely different; in particular, we will be able to solve the remaining task of finding a smaller feedback vertex set that is disjoint from the given one in polynomial time, whereas Chen et al. [33] still require exponential (in $k$) time for this task in the case of Feedback Vertex Set on general directed graphs, as well as Dehne et al. [36] and Guo et al. [35] when solving.

Consider a FVST Disjoint Compression instance $(T, S)$. As mentioned, both $T \setminus S$ and $T \setminus V \setminus S$ are acyclic and thus have a topological sort. Then, the topological sort of a maximum acyclic sub tournament of $T$ containing all of $S$ can be thought of as result from inserting a subset of $V \setminus S$ into the topological sort of $S$.

**IV. IMPLEMENTATION DETAILS**

**A. Language used.**

The algorithms were implemented using python language.

**Python** : Python is an interpreted, interactive, general-purpose programming language used for high-level programming.[39] Python claims to combine "remarkable power with very clear syntax" and its standard library is large and comprehensive.

Python supports multiple programming paradigms, primarily but not limited to object-oriented, imperative and, to a lesser extent, functional programming styles. It features a fully dynamic type system and automatic memory management, similar to that of Scheme, Ruby, Perl, and Tcl. Like other dynamic languages, Python is often used as a scripting language, but is also used in a wide range of non-scripting contexts.

**B. Graph Library**

The Graph library used is networkx. NetworkX is a Python library for studying graphs and networks. NetworkX is free software released under the BSD-new license. The features of networkx library are as follows:

1. Huge no of standard graph algorithms.
2. Well documented and tested
3. Generator for random graphs, symmetric graphs.
4. Classes for graphs and digraphs.
5. Conversion of graphs to and from several formats.
6. Ability to find subgraphs, cliques, k-cores.
7. Ability to find subgraphs, cliques, k-cores.
8. Explore adjacency, degree, diameter, radius, center, betweenness, etc.
9. Draw networks in 2D and 3D.
C. Library for plotting

Matplotlib.pyplot is used for plotting the results. matplotlib.pyplot is a collection of command style functions that make matplotlib work like MATLAB. Each pyplot function makes some change to a figure, e.g: create a figure, create a plotting area in a figure, plot some lines in a plotting area, decorate the plot with labels, etc. This library is used to draw plots for results obtained by the spanner algorithms.

Input test cases

The efficiency of the algorithms can be best judged when the input graphs are dense. So, using a python program, complete graphs are generated with varying number of vertices. The range of number of vertices is [100,500]. All the graphs are unweighted and undirected. The algorithm runs efficiently even the number of edges in the graph around 2000 edges.

Input Graph representation:
The input graph is to be taken as edge list, the specified input file should contain the number of vertices in the first line, followed by the edge list, one edge per line. The algorithm reads edge by edge. The following is an example input file:

```
4
1 3
2 0
3 2
```

D. Data Structures

1. Graphs: networkx graph library provides in-built Graph data-structure. This graph data-structure is used to graphs.
2. Queues: Priority Queues are used in our implementation for topological sort for hitting sets. These Queues uses dynamic memory allocation.
3. Lists: list is sequential collection of python objects : numbers, strings functions, objects, and even other lists. List are used in our implementation to store results, and in sorting edges based on their weights etc.
4. Dictionaries: Dictionary in Python is a data type also known as map. Each element of a map is accessed (or, indexed) by an unique key, and so they are known as key-value pairs. Python inbuilt dictionary data structures are used in our implementation to store distance in formation of each vertex from source, and also shortest path trees from a source vertex.
5. Sets: A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

E. Functions Description

This section explains the functions used in spanner algorithms implementation.

In-built networkx functions

The following in-built networkx library functions are used in spanner algorithm implementation.

Algorithm for Odd Cycle Transversal:

Table 5.1: Run times in seconds for different datasets

<table>
<thead>
<tr>
<th>No of vertices</th>
<th>no of edges</th>
<th>Parameter k</th>
<th>Timetake</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>10</td>
<td>4</td>
<td>0.023</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
<td>4</td>
<td>0.12</td>
</tr>
<tr>
<td>167</td>
<td>641</td>
<td>10</td>
<td>139</td>
</tr>
<tr>
<td>167</td>
<td>641</td>
<td>16</td>
<td>268.88</td>
</tr>
<tr>
<td>250</td>
<td>400</td>
<td>10</td>
<td>577.93</td>
</tr>
<tr>
<td>250</td>
<td>400</td>
<td>15</td>
<td>2192</td>
</tr>
<tr>
<td>500</td>
<td>750</td>
<td>10</td>
<td>312.23</td>
</tr>
<tr>
<td>500</td>
<td>750</td>
<td>15</td>
<td>3266.32</td>
</tr>
<tr>
<td>500</td>
<td>750</td>
<td>25</td>
<td>7669.12</td>
</tr>
</tbody>
</table>
A. Observations

1. Over 90% of the time is spent in finding an augmenting path within the flow network; all that this requires from a graph data structure is enumerating the neighbors of a given vertex.

2. Finding Vertex Cuts. It has now become clear that in the inner loop of the algorithm, we need to find a minimum vertex cut between two sets $Y_1$ and $Y_2$ in a graph $G$, or equivalently, a maximum set of vertex-disjoint paths between two sets. This is a classical application for maximum flow techniques: The well-known max-flow min-cut theorem tells us that the size of a minimum edge cut is equal to the maximum flow. Since we are interested in vertex cuts, we create a new, directed graph $G$ for our input graph $G = (V, E)$: for each vertex $v \in V$, create two vertices $v_{in}$ and $v_{out}$ and a directed edge $(v_{in} \rightarrow v_{out})$. For each edge $(v, w) \in E$, we add two directed edges $(v_{out} \rightarrow w_{in})$ and $(w_{out} \rightarrow v_{in})$. It is not too hard to see that a maximum flow in $G$ between $Y_1 := y_{in} \in Y_1$ and $Y_2 := y_{out} \in Y_2$ corresponds to maximum set of vertex disjoint paths between $Y_1$ and $Y_2$. Furthermore, an edge cut $D$ between $Y_1$ and $Y_2$ is of the form $v \in V$ $(v_{out} \rightarrow v_{in})$, and $(v_{in}, v_{out})$ is a vertex cut between $Y_1$ and $Y_2$ in $G$. Since we know that the cut is relatively small (less than or equal $k$), we employ the Edmonds-Karp algorithm. This algorithm repeatedly finds a shortest augmenting path in the flow network and increases the flow along it, until no further increase is possible.

B. Algorithm for Feedback Vertex sets

Table 5.2: Run times in seconds for different datasets

<table>
<thead>
<tr>
<th>No of vertices</th>
<th>$k=2$</th>
<th>$k=4$</th>
<th>$k=6$</th>
<th>$k=8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>0.20</td>
<td>0.83</td>
<td>18.23</td>
<td>359.13</td>
</tr>
<tr>
<td>60</td>
<td>0.58</td>
<td>2.45</td>
<td>18.15</td>
<td>1222.10</td>
</tr>
<tr>
<td>80</td>
<td>0.97</td>
<td>5.14</td>
<td>35.51</td>
<td>386.83</td>
</tr>
</tbody>
</table>

1. The algorithms performance scales poorly with an increase of the parameter $k$, which is typical for FPT algorithms. For $k = 8$ and $ed = 2.0$, the algorithm never finished within 3 hours. For $ed = 3.0$ and $k = 8$, the algorithm began to time-out when $n$ is larger than 140.

2. The Impact of the Parameter $k$ : When the algorithm is run with varying parameter from 2,4,6,8, the runtime reaches its peak when the parameter $k$ is near the size of the minimum feedback vertex set. The runtime is quite fast when $k$ is much larger or much smaller than the optimum.

3. The runtime is not monotonely increasing in $n$. The fluctuations of the curves seem to indicate that the runtime is dominated by the exponential dependency on $k$. For example, the high runtime for $(40, 8, 2.0)$ is due to the fact that these graphs have many independent cycles, and Chens algorithm seems to perform poorly on such graphs. A possible explanation may be found in the iterative compression technique.

4. The performance does not increase monotonely in the edge density. The algorithm can still quickly solves all instances with small parameter $k$ when the number of edges increases significantly. Also, for bigger $k$, the runtime does not increase monotonely with the number of edges. For example, for $k = 8$, the problem becomes most difficult for the FPT algorithm when $ed = 2$. 

![Figure 5.1: Graph for OCT algorithm for various parameters](image1)

![Figure 5.2: Graph for algorithm for various parameters](image2)
VI. CONCLUSION AND FUTURE WORK

We have presented the importance of Fixed parameter Tractability, to solve the NP-hard problems efficiently. We have studied the different case studies for fixed parameter algorithms, mainly concentrating on Graph Bipartization. Survey on Odd Cycle Transversals and Feedback vertex sets are done and recent algorithms to solve these problems are implemented and results are analyzed based on the different parameters on a varied datasets.

Techniques like Iterative Compression and Kernalization are to be implemented extensively to the remaining sets of NP-hard problems. Iterative compression can also be employed to compress a non-optimal solution until an optimal one is found. Some reduction rules can be studied and applied to these algorithms for making the implementation of these NP-hard problems very practical.

REFERENCES

[28] P. Sasatte, Improved FPT algorithm for feedback vertex set problem in bipartite
[37] Michael Dom a, Jiong Guo c, Falk Hffner b, Rolf Niedermeier a, Anke Truss Fixed-parameter tractability results for feedback set problems in tournaments 2009
[38] //snap.stanford.edu/data/