Rt
407
c.1

Fixed-Parameter Algorithms for Hitting Set Problems

by

Rola El Masri

B.S., Computer Science, Lebanese American University, 1996

Thesis submitted in partial fulfillment of the requirements for the Degree of Master of Science in Computer Science

Division of Computer Science and Mathematics

LEBANESE AMERICAN UNIVERSITY

June 2006

LEBANESE AMERICAN UNIVERSITY

School of Arts and Sciences - Beirut

Thesis approval Form (Annex III)

Student Name: Rola S. Masri I.D. #: 199308260

Thesis Title : Fixed Parameter Algorithms for Hitting set Problems

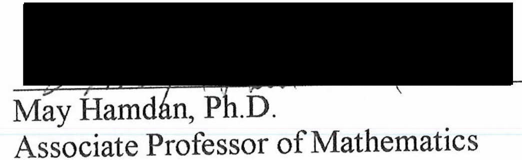
Program : Computer Science

Division/Dept : Computer Science and Mathematics

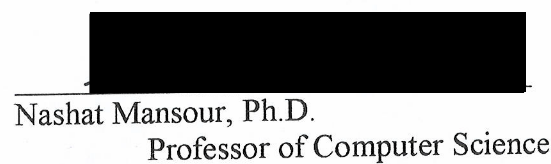
School : Arts and Sciences - Beirut

Approved by :

Faisal N. Abu Khzam, Ph.D. (Advisor)
Assistant Professor of Computer Science



May Hamdan, Ph.D.
Associate Professor of Mathematics



Nashat Mansour, Ph.D.
Professor of Computer Science

Date : June 16, 2006

Plagiarism Policy Compliance Statement

I certify that I have read and understood LAU's Plagiarism Policy. I understand that failure to comply with this Policy can lead to academic and disciplinary actions against me.

This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that by acknowledging its sources.

Name: Rola S. Masri

Signature:

Date: 29/6/2006

I grant to the LEBANESE AMERCIAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or its students and employees. I further agree that the University may reproduce and provide single copies of the work to the public for the cost of reproduction.

To my parents

Acknowledgment

First and foremost, I would like to express my profound gratitude to my advisor Dr. Faisal Abu Khzam who was very helpful in giving me continuous support, constant encouragement and efficient advice. I want to thank also Dr. Nashaat Mansour and Dr. May Hamdan for being on my thesis committee.

I want to express my sincere gratitude to my lovely husband Ahmad who endured with me a lot. Thank you darling for all the love, care, and support you gave me. Thank you for taking care of our two wonderful daughters at night and taking them out sometimes to let me focus on my studies. Thank you for being with me to persevere during the bad times and celebrate the good times.

I would also like to thank my sister Faten for providing me with much needed moral support along the way. Thank you sister for always calling me from China to give me encouragements and strength. Thank you for having confidence in me. You have never doubted that I would complete this thesis in this semester while I certainly had my doubts from time to time. You used to always tell me: "you can do it" and I did it.

I would also like to thank the Arab Bank and the Lebanese American University whose financial supports during my graduate studies made it all possible.

Finally, I want to express my deepest appreciation to my lovely parents for their unlimited support. They taught me the importance of education and always encouraged me to work harder and better. Thank you dad for your precious sacrifices and your continuous encouragement and faith in me. Thank you for teaching me how to be ambitious and never give up. Thank you mum for being so patient and always willing to help me without complaining or nagging. Thank you for taking care of my two little daughters while I was studying. To you I owe the greatest moral debt that cannot be repaid. I am what I am because of you and your prayers. To you I dedicate this thesis.

Abstract

The Hitting Set problem received a great deal of attention lately due to its plethora of applications in scientific domains. Many Algorithms have been developed to solve this problem by targeting the reduction of the worst-case run time. The most recent proposed algorithm employs a technique called pseudo-kernelization, introduced by F. Abu Khzam. Pseudo-kernelization seeks to find favorable conditions whose presence leads to a faster search strategy, and whose absence leads to a better reduction of the problem instance. When applied to the 3-Hitting Set problem, it provides a great improvement over the previously known algorithms. In this paper, we consider the algorithm of H. Fernau, which has the best worst-case run time. We show that combining the methods used in Fernau's algorithm and the pseudo-kernelization technique gives the best run times for 3-Hitting Set. We provide empirical evidence that supports this claim. Moreover, we show how applying pseudo-kernelization to the general d-Hitting Set problem provides an improvement over the previously developed algorithms.

Contents

1	**Introduction**	1
2	**Preliminaries and background**	4
3	**Applications for Hitting Set Problem**	7
	3.1 Placement Strategy	7
	3.2 Covering Trains by Stations	9
	3.3 Hitting Set Algorithm for Gene Expression Analysis	9
	3.4 Interference in Cellular Networks: A Hitting set problem	10
4	**Fixed-Parameter Algorithms: The Bounded Search Tree Technique**	12
	4.1 Kernelization	13
	4.2 Bounded search tree	16
	4.3 Interleaving	21
	4.4 Pseudo-Kernelization	22
5	**3-Hitting Set: Recent Bounded Search Tree Algorithms**	24
	5.1 Kernelization	25
	5.2 The bounded Search Tree	26
	5.2.1 Bounded Search Tree Algorithm by Niedermeier and Rossmanith	26
	5.2.2 Bounded Search Tree Algorithm by Henning Fernau	27
	5.2.2.1 General Notions and Definitions	27
	5.2.2.2 Henning Fernau Bounded Search Tree Algorithm	28
	5.3 Interleaving of 3HS Problem	31
	5.4 Bounded search Tree of Niedermeier and Rossmanith compared to that of Fernau	31
	5.5 Pseudo-Kernelization of 3HS Problem	32
	5.5.1 Quadratic Pseudo-Kernel of 3HS Problem	32
6	**d-Hitting set and Pseudo-Kernelization**	34
	6.1 Bounded Search Tree for d-Hitting Set Problem	34

6.2 Pseudo-Kernelization for d-Hitting Set problem 35

7 **An Improved Algorithm for 3-Hitting Set Problem** 39

7.1 An Improved Algorithm for 3-Hitting Set Problem: A Detailed Description 39

8 **Experimental Results** 41

8.1 Implementation Details 41

8.2 Random Generator 42

8.3 Statistics and Conclusion 43

9 **Conclusion** 54

10 **References** 56

Appendix Technical Details 58

List Of Figures:

Figure 3.1 The art gallery problem as set cover problem. 8

Figure 3.2 The art gallery problem as hitting set problem. 9

Figure 4.1 The pseudo-code of Buss' kernelization. 16

Figure 4.2 The bounded search tree of the Vertex Cover problem. 20

Figure 4.3 The pseudo-code of the bounded search tree of the Vertex Cover. 20

Figure 5.1 3-Hitting-Set example. 25

Figure 8.1 - The pseudo-code of Fernau Branching Technique. 44

Figure 8.2 - The pseudo-code of "Reduction_rules". 45

Figure 8.3 - The pseudo-code of "Heuristic_priorities". 46

Figure 8.4 - The pseudo-code of the Pseudo-Kernelization Technique. 47

Figure 8.5 - The pseudo-code of "Check_Pseudo_Kernelization". 49

Figure 8.6 - The pseudo-code of "Pseudo_Kernel reduction". 50

Figure 8.7 - The pseudo-code of "Pseudo_Kernel branch". 51

Figure 8.8 - The pseudo-code of the Random Generator. 52

List Of Tables:

Table 4.1 The upperbounds of different algorithms for solving vertex cover problem. 21

Table 5.1 The different cases considered in when there is one subset of size two. 30

Table 5.2 The different cases considered when there are two subsets of size two. 30

Table 5.3 Comparison between the Algorithm of Niedermeier and Rossmanith and that of Fernau. 32

Table 6.1 Results of T(k) of the algorithm of Niedermeier and Rossmanith for solving d-hitting set problem. 35

Table 6.2 Results of T(k) of the Pseudo-Kernelization algorithm for solving d-hitting set problem. 36

Table 8.1 Experimental results of the pure branching and hybrid algorithm. 53

Chapter 1

Introduction

The complexity class NP contains many problems that people would like to be able to solve efficiently because they model real life problems. The necessity to solve these problems has created a variety of different methodologies on how to cope with these intractable problems. One of the most recent methodologies is due to the emergence of *Fixed-Parameter algorithms.*

When trying to solve NP-hard problems exactly, we have to deal with exponential running times. This is believed to be inevitable for such problems. Parameterized complexity proposes another perspective in measuring the difficulty of intractable problems. Its main slogan is: not all NP-hard problems are equal. The main concept is to consider some input parameter, associated with a problem in hand, that could be responsible for the exponential growth in the run time of some algorithmic solution. Such exact methods are nowadays called *Fixed-parameter algorithms*. Thus, if the said parameter is fixed, the problem becomes tractable.

Formally, a parameterized problem is one whose input is of the form (X, k), where k is a specific input instance of total size n and k is an input parameter. To give a concrete example of what a parameter could be, we note that almost all such problems have the target solution size as their parameter.

One of the most important hard problems is the *Hitting Set problem*. Formally, the Hitting Set problem is defined as follows:

Given: a collection of subsets C of a finite set S and a positive integer k.

Question: Can we find a subset H of S with $|H| \leq k$ such that H contains at least one element of each subset in C.

The Hitting Set problem is called d-Hitting Set if every subset in C is of size not exceeding a fixed positive integer d. The vertex cover problem can be posed as a 2-Hitting Set problem. 3-Hitting Set can be seen as a Vertex Cover problem for hypergraphs. The hyperedges in hypergraphs are between three vertices instead of two vertices. The problem is to find the minimum subset of vertices that covers all hyperedges.

Two algorithms arose lately to solve the Hitting Set problem in time that is exponential in the parameter k. These algorithms are: The bounded search tree algorithm of Niedermeier & Rossmanith [4] and that of Fernau [3]. In the former, the bounded search tree works in two stages: Kernelization and Bounded search tree. Kernelization reduces the input instance to an equivalent smaller instance and bounded search tree works on the reduced instance and solves the problem by traversing a search tree. When applied to 3-hitting set, kernelization reduces the instance to a size cubic in the parameter k and the algorithm runs in $O(2.270^k + n)$. In the latter, the branching algorithm works by applying reduction rules and heuristic priorities. Reduction rules aim at reducing the input instance to a smaller instance and the heuristic priorities consider priorities for choosing an element on which the branching is favorable. When applied to the 3-hitting set problem, the branching algorithm runs in $O(2.1788^k + n)$.

Pseudo-kernelization, which is a new technique introduced by Dr. Faisal Abu Khzam [2], did a great improvement on the algorithms introduced lately. When applied to 3-hitting set problem, it gives either better run time which is $O(2.05^k + n)$ or smaller reduced instance, whose size is quadratic in the parameter k and could also deliver instances whose size is linear.

In this thesis, we use the methods of Dr. Faisal Abu Khzam and apply the pseudo-kernelization technique to the d-Hitting Set problem. This is done by first applying the

reduction rules of [3] to the given instance and then by considering a favorable condition whose presence results in a better run time compared to the previously developed algorithms and whose absence results in a smaller kernel instance.

Also in this paper, we present an algorithm that has proved to give the best results for the 3-Hitting Set Problem. It is a hybrid algorithm that combines the heuristic of H. Fernau Algorithm and pseudo-kernelization. We highlight the practical efficiency, efficacy and usage of this improved algorithm by implementing and testing our algorithm and then comparing its results with those of Fernau [3].

The chapters are organized as follows:

Chapter 2 explains basic background material. Chapter 3 highlights some of the applications of the Hitting Set problems. These applications range from Biology to Networking. Chapter 4 gives a detailed description of the bounded search tree technique and kernelization technique, using the vertex cover problem as an exemplar. Chapter 5 shows how the bounded search tree and the pseudo-kernelization techniques apply to the 3-Hitting Set problem. Chapter 6 extends the 3-Hitting Set pseudo-kernelization to the d-Hitting Set problem. Chapter 7 provides a detailed explanation of our improved 3-Hitting Set algorithm. Chapter 8 presents the implementation and the experimental results. Finally, chapter 9 is devoted for a brief summary and conclusions.

Chapter 2

Definitions and Preliminaries

The purpose of this chapter is to explain basic definitions and some concepts necessary to understand this paper. We review basic definitions and concepts regarding algorithm.

2.1 Computational Complexity Classes

Computational complexity theory is concerned with the existence of efficient algorithms, while Computational theory is concerned with the existence of algorithms for a problem.

The complexity class P is the set of problems that can be solved in polynomial time by a deterministic machine. The complexity class NP is the set of problems that can be solved in polynomial time by a non-deterministic machine. This class contains many problems that many researchers would like to be able to solve as efficiently as possible. Of particular interest in our work is the Hitting Set problem, which falls in the class NP, but believed not to belong to P.

The complexity class NP-complete is the set of problems that are the hardest problems in NP. If one could find an algorithm to solve an NP-complete problem quickly, then surely one could use that algorithm to solve all NP problems. The complexity class NP-hard is the set of problems that are at least as hard as any problem in NP.

2.2 Parameterized Problems and Fixed-Parameter Tractable (FPT) Problems

As mentioned earlier, parameterized problems have the following general form: given an input X and a nonnegative integer k, does X have some property that depends on k? The parameter k is considered usually to be small when compared to the input size. The question is whether we can find an algorithm which is exponential in k and not in the

input size. A problem is fixed-parameter tractable if it can solved in time $O(f(k) * |x|^{O(1)})$ where f is a function that depends on k. The complexity class of Fixed-parameter tractable problems is called FPT.

2.3 Vertex cover and Hitting Set

A vertex cover of an undirected graph G = (V, E) is a subset V' of the vertices of the graph V that contains at least one of the two endpoints of each edge. This problem is equivalent to 2-Hitting Set. To see this, note that V can be treated as the set S and E as the set C since edges of a graph can be represented as sets of size two and a vertex cover of G is a Hitting Set of (V, E).

2.4 Other useful definitions

Definition 1.In a collection of subsets C, we say that {x, y} co-occur in n subsets if these n subsets contain x and y at the same time.

Definition 2. Given a collection C of a finite set S. If each $x \in S$ occurs exactly in d subsets of the given collection C, we say that the collection is d-regular.

Definition 3. The Set Cover problem is defined as follows:
Given a collection $C = \{C_1, C_2, \ldots, C_n\}$ of subsets of a finite set S. The set cover problem is to find a subset $H \subseteq C$, such that H covers all elements in S and |H| is minimum.

Definition 4. Hyperedge and hypergraph are defined as follows:
When an edge is connecting two vertices, we call it simply an edge. But when an edge is connecting more than 2 vertices, we call it hyperedge. A graph containing hyperedges is called hypergraph.

Definition 5. A vertex-induced subgraph is one that consists of some of the vertices of the original graph and all of the edges that connect them in the original. An edge-induced

subgraph consists of some of the edges of the original graph and the vertices that are at their endpoint

Chapter 3

Applications for Hitting Set Problem

3.1 Placement Strategy

One of the most important tasks for a mobile robot is to depict a representation of the environment using its visual sensors. These visual sensors have to be put throughout the workspace in order to get this representation. An important question is: Which locations should the robot visit in order to gather the images necessary to represent the whole workspace? How many sensors, in general, should be placed to build the image of the workspace as efficiently as possible?

A robot can capture a 3-D representation of the workspace at a particular location by performing rotational sweep and obtaining several 3-D image of the workspace during the sweep. The 3-D image consists of a set of points and its resolution is inversely proportional to the speed of the rotational sweep. A slow sweep produces high resolution 3-D image. As a result, the acquisition of a 3-D image is a costly operation.

A placement strategy is a strategy to determine a set of good locations where visual sensors will be the most effective in depicting a representation of the whole workspace. In the case of robot, the problem is to find the minimum set of locations that the robot should visit such that the whole workspace is represented as 3-D image. In fact, assuming that a polygonal 2-D map of the workspace is available, the problem is now to find the smallest set of locations in the 2-D map from which the entire contour is visible. The robot then visits these locations to acquire the 3-D image of the workspace.

The sensor placement strategy is a variant of the art-gallery problem: Find the minimum set of guards from which any point of the art gallery is visible. The simple art-gallery

problem is NP hard and it can be transformed into a hitting set problem or a set cover problem.

The art gallery can be transformed into a set cover problem as follows: Let $G = \{g_1, g_2, ..\}$ be the large set of m locations selected at random in the workspace W. Let ∂W be the boundary of the workspace W. for every edge in the workspace $e \in \partial W$, the edge is decomposed into cells. The points in the same cell are visible by a subset of G. the cells are enumerated and group under the set $X = \{1,2,\ldots,l\}$ l is the number of cells. Let $R = \{R_1, R_2, \ldots, R_m\}$ where $R_i = \{x \in X \mid g_i \text{ covers } x\}$ is the subset of element of X that are covered by g_i. The union of $R_i = X$. the problem is now to find the minimum set cover $C \subseteq R$ such that the union of all R_i in $C = X$. The cardinality of C is the number of subsets in C. Figure 3.1 shows the art gallery problem as a set cover problem.

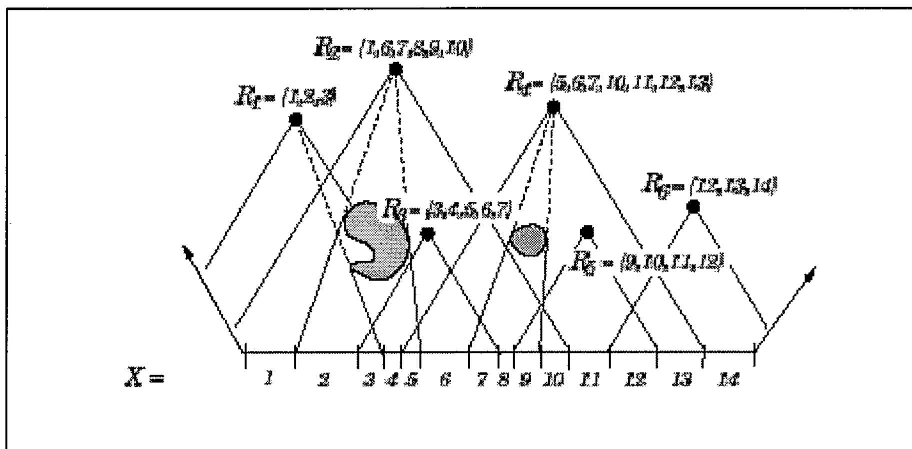


Figure 3.1 - The art gallery problem as set cover problem

The art gallery problem can be transformed into a hitting set cover as follows: Let X' be the large set of guard candidates in the workspace. Let $R' = \{R_x \mid x \in X\}$ where R_x consists of all subsets $R \in$ R that contains x. The problem is now to find a subset $H' \subseteq X'$ such that $H' \cap X' \neq \emptyset$. In other words, the problem is to find the minimum set of guards that hits all the subsets $R' \in R'$. Figure 3.2 shows the art gallery problem as a hitting set problem

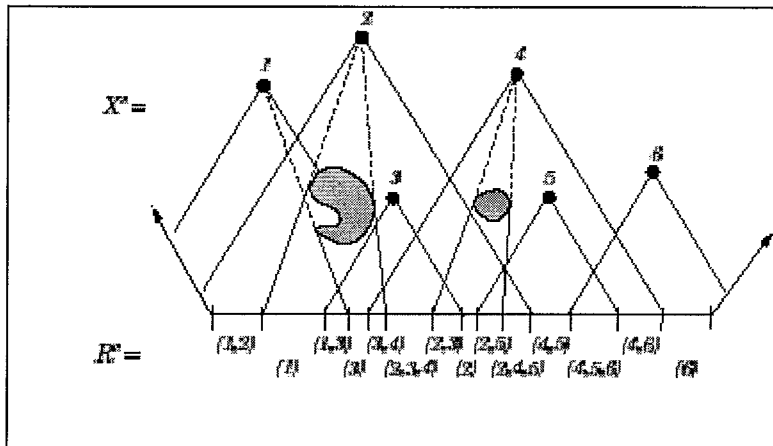


Figure 3.2 - The art gallery problem as hitting set problem

3.2 Covering Trains By Stations

"Covering trains by stations" was introduced by Karsten Weihe. Given a finite number of trains, find the minimum set of stations such that every train meets at least one of these stations. This problem is a hitting set problem and can be explained as follows: Let S be a set of stations by which at least one train passes. Let C be equal to $\{V_1, V_2 \ldots V_m\}$ such that $\forall\ V_i \in C$, V_i is a subset of S such that all stations in V_i are met by one train. Hence, the problem is to find a subset $V' \subseteq V$ such that $V' \cap V_i \neq \emptyset$.

3.3 Hitting Set Algorithm for Gene Expression Analysis

In Biology, the gene expression of n genes can be measured in the same experiment. In such experiments, the changes in gene expression of thousands of genes are observed over a period of time. A genetic network is a model that shows how the expression level of a gene is affected by the expression level of other genes. Given the results of an experiment with n genes and m measures, the problem is to find a subset of the set of genes that affects the expression level of a given gene.

Let $E = \{a_0, a_1, \ldots, a_{n-1}\}$ be the set of the n genes under study. Let m be the number of measurements in which the expression levels of the n genes are observed. Hence, the input of the algorithm is a matrix M of size n*m of values 1 or 0. Value 1 means

expression while value 0 means non-expression. Let a_r be the target gene for which we want to find the genes that are responsible of its different expression levels.

As an example consider the matrix shown below:

M=

a_0	a_1	a_2	a_3	
1	1	1	0	0
0	1	0	1	1
1	0	0	0	2
1	1	0	1	3

In this example, n = 4 and m = 4 (number of rows). Let a_3 be the gene under study. The values of a_3 are different in rows (0,1), (0,3), (1,2) and (2,3). The expression levels of the genes a0 and a2 show different value in row 0 and 1. Hence, $S_{01}= \{a_0,a_2\}$. If we apply the same observation, we get $S_{03} = \{a_2\}$, $S_{12} = \{a_0,a_1\}$ and $S_{23} = \{a_1\}$. The problem is to find the smallest set of genes that explain the differences of expression levels of a_3. In other words, we want to find the smallest HS that hits all S_{ij} for all row pairs (i,j). Here, the genes that affect the expression levels of a_3 are $H = \{a_1,a_2\}$.

In general, the gene expression levels can be transformed into a Hitting-Set problem as follows: given E the set of genes and a finite collection $S= \{S_1,\ldots,S_w\}$ of subsets of E, find a subset $H \subseteq E$ such that $\forall S_i \in S, H \cap S \neq \emptyset$.

3.4 Interference in Cellular Networks: A Hitting set problem

Cellular networks consist mainly of two nodes: Base stations acting as servers and clients. Interference can occur at a client if it lies within the transmission range of more than one server. In order to minimize interference, every base station must be assigned a transmission power level such that a client is covered by a minimum number of base

stations. The interference problem can be presented as a set cover problem or hitting set problem. The task of reducing interference is formalized as set cover problem as follows: Given a set of elements U which is the set of all clients covered by all stations. Let S be a collection of subsets of U such that each subset represents the transmission range of a base station as a set containing exactly all clients covered thereby. The problem is to find a subset $S' \subseteq S$ such that every client occurs in at least one set of S'.

On the other hand, the interference problem can be represented as hitting set problem as follows: Let U be a finite set of all base stations and let S be a collection of subsets of U such that for $\forall S_i \in S$, Si contains all the base stations that contain a client c. the problem is to find a subset $U' \subseteq U$ such that for $\forall S_i \in S$, $U' \cap S_i \neq \emptyset$. in other words, we have to find the minimum number of base stations that cover all clients. In order to minimize interference, every base station in U' is then assigned a transmission power level.

Chapter 4

Fixed-Parameter Algorithms: The Bounded Search Tree Technique

Many algorithmic techniques have been introduced to solve parameterized problems. The main aim of these algorithms is to "find the fastest FPT algorithm for the problem and to find the smallest polynomial time computable Kernelization (reduced instance) for the problem" [10]. In this chapter, we explain three techniques that are the most common to solve FPT problems. These techniques are kernelization, bounded search tree (also known as branching) and interleaving. Kernelization is a pre-processing stage in which the input instance is transformed into an equivalent one, called a kernel instance, whose size is bounded above by a function of the parameter only. In the bounded search tree phase, the kernel instance is solved via recursive backtracking in a manner that guarantees reduction of the parameter as the search goes from one search-tree level to another lower one. Since the parameters are getting smaller in the recursive calls, the recursion terminates either by finding a solution bounded above/below by the parameter or it halts with no solution. Interleaving is simply the re-application of kernelization at every node of the search tree. A detailed explanation of each technique will be given in the following sections.

In addition, a new technique, called pseudo-kernelization, will be discussed. This new technique is used to improve FPT algorithms namely those that employ the bounded search tree and kernelization techniques. A detailed description of pseudo-kernelization will follow in section 4.4.

4.1 Kernelization

When looking up the meaning of Kernel in the dictionary, it means the important or main part of something, often surrounded by unimportant or untrue matter. The question is: what is the meaning of Kernel in Computational Complexity Theory? It is interesting to know that Kernel has exactly the same meaning as that in the dictionary. In fact, the process of producing a problem kernel is nothing but a polynomial-time reduction of an input instance (G, k) into a smaller and equivalent instance (G', k'). Hence, when kernelization is applied to any FPT problem we are left with the important and main part of the problem instance, the core instance.

Kernelization is an indispensable pre-processing step for any problem on which it can be applied: since the search for a solution takes exponential time, then surely dealing with the smaller problem kernel can make the problem easier to solve.

Kernelization satisfies the following:

1- (G, k) is a yes-instance if and only if (G', k') is a yes instance. Hence, kernelization removes elements from the original problem instance without affecting its solution.
2- The size of the reduced problem instance is bounded above by some function g of the parameter k (g is independent of $|x|$).
3- The transformation of the original instance problem (G, k) into the smaller equivalent instance problem is performed in polynomial time. In other words, we are interested in achieving g(k) as small as possible in polynomial time[10].
4- The reduction of a problem instance (G, k) into a smaller instance may sometimes result in reducing the parameter k. in other words, $k' \leq k$.

Note that g is an arbitrary function depending on k only. If g is a linear function, we say that we have linear kernel and if g is quadratic function, we say that we have quadratic kernel, and so on.

There are four essential requirements when designing a Kernelization algorithm:

1- Time Efficiency: The reduced (kernel) instance must be obtained by applying the most efficient polynomial time algorithm. Otherwise, interleaving the kernelization steps would not be assailable.
2- Practicality: The size of the reduced instance must be small to the extent of being easy to use in practice. Kernelization "reduces a problem so that often a brute force search is tractable on the resulting kernel" [11]. There are problems that may be reduced to cubic size of the parameter k. However, it is worth mentioning that when applying other technique called pseudo-kernelization (the Pseudo-Kernelization technique is discussed in section 4.4), the problem instance can be reduced to quadratic and even linear size of the parameter k.
3- Constructability: We must be able to use the solution of the kernel instance in obtaining a solution of the original instance (in polynomial time).
4- Flexibility: Kernelization can be applied not only in the preprocessing phase but also in the bounded search tree phase (Bounded search tree technique is discussed in the following section). Note that some algorithms "use kernelization as a merely pre-processing stage (such as Niedermeier & Rossmanith 1999 for the vertex cover problem) whereas other algorithms apply kernelization after every round of the algorithm (such as Downey et al.1997 also for the vertex cover problem)" [11].

Reduction to a problem kernel is done through the application of reduction rules that vary from problem to another. Most of the problems are reduced by applying a set of reduction rules and not a single reduction rule. Usually, each reduction rule reduces the input instance by removing elements that may or may not be added to the solution. Take as an example the 3-Hitting Set kernelization of Niedermeier and Rossmanith that will be discussed in detail in chapter 5. It is done by applying two rules. In fact, applying the first rule may remove elements that will not be added to the solution, whereas applying the second rule may remove elements that will be added to the solution. However, there are

problems that are reduced by applying a single reduction rule, but these are rare. The original reduction rule of vertex cover from Buss' algorithm (Buss & Goldsmith 1993) is an exemplar.

Actually the most cited application of kernelization is the simple quadratic kernel of Vertex Cover from Buss' algorithm (Buss & Goldsmith 1993), which we include here for illustration. Recall that, for a given Vertex Cover instance (G, k), the question is to find a set of k (or less) vertices whose removal deletes all edges.

A quadratic problem kernel for VC can be obtained by selecting all the vertices that have a degree $\geq$ k. let p be the number of these vertices. If p $\geq$ k then we deduce that there is no vertex cover of size at most k. Otherwise, the new parameter k' will be equal to k – p. Then, we delete the p vertices together with all their incident edges. If the number of the remaining edges is greater than k * k' then no vertex cover of size less than or equal to k exists. Figure 4.1 represents the pseudo-code of the vertex cover algorithm of Buss. Obviously the new instance is asymptotically bounded by O (k^2).

Many algorithms have been proposed to reduce further the kernel instance of the vertex cover problem. Most of these algorithms include the kernelization rule of Buss' algorithm and some other rules. All in all, there are mainly four rules mentioned in [11] that can be identified from many different algorithms to reduce a vertex cover problem into its problem kernel. The four rules are:

1- All the vertices $\{v_1, v_2, v_3, \ldots, v_p\}$ that have degree greater than k should be included in the vertex cover and deleted from the graph along with all their incidents edges. The reduced instance is (G', k') with G' = G - $\{v_1, v_2, v_3, \ldots, v_p\}$ and k'= k - p.
2- If a vertex u is of degree one and has an edge with v, then v can be in the vertex cover. The reduced instance is (G', k') with G' = G - {u, v} and k'= k - 1.

3- If a vertex u has only two adjacent neighbors, y and z, then y and z can be placed in the vertex cover. The reduced instance is (G', k') with G' = G - {y, z} and k'= k - 2.

4- If two vertices u and v are adjacent such as N (v) $\subseteq$ N (u), then u should be in the vertex cover. The reduced instance is (G', k') with G' = G - {u} and k'= k – 1.

Input: A graph G = (V; E) and an integer k (n = |V |; m = |E|).

Repeat

  Select a vertex v $\in$ |V| edge such as v has degree $\geq$ k

  Compute G' = G [V – v]

  Compute k' = k - 1

Until there are no vertices that have degree $\geq$ k

If m > k* k' then

  Return "No solution"

Else

  Return (G', k')

Figure 4.1 - The pseudo-code of the of Buss' kernelization.

4.2 Bounded Search Tree

Many combinatorial problems can be solved trivially using the Brute Force technique (one that looks at all possible subsets of size k). The size of the search tree is hence exponential. "The exponential worst-case complexity of such algorithms comes from problem instances where we need complete tree traversal" [13].

However, by considering parameterized problems, the size of the search tree is now bounded by function of the parameter k and is called Bounded Search Tree. Most of the FPT problems can be solved using the bounded search tree technique, also known as

"Branching." In the realm of fixed-parameter algorithms, branching consists of using the parameter as a guide in clever traversals of the bounded search tree.

Bounded search tree algorithms are recursive. The search tree is constructed by choosing any instance at the beginning as the root of the tree and then recursively applying a branching rule that creates children of the root node. Each node of the tree corresponds to an instance of a graph.

The main focus of the bounded search tree techniques is to minimize as much as possible the maximum number of children of a node and to limit the height of the search tree:

1- "The height of the tree represents the maximum number of choices we are allowed to make before we must be able to determine a solution" [12]. In the case of minimization FPT problems, we choose one or more solution to the problem while moving from one level of the search tree to another, thus decreasing the parameter k. This creates a tree of size bounded above by the parameter k and where one path from the root to a leaf may lead to a solution.
2- The maximum number of children of a node depends mainly on the branching rule that is used. It is mainly the branching rule that draws the structure of the search tree. In what follows, two methods of the bounded search tree for the vertex cover problem are described. These two methods use different branching rules. A close understanding of these two methods shows how the difference in branching rule leads to minimize the number of children of a node.

Taking the above into consideration, bounded search tree has the following properties:

1- The height of the search tree depends only on the parameter k, which is fixed.
2- The algorithm is of polynomial time.
3- The algorithm is recursive.

4- The maximum number of children of a node depends on the branching rule that is used to branch on.

The most cited application for this technique is for the vertex cover problem. A Brute-Force algorithm takes time of O(n^k). Can we do better?

Consider the following two ways for constructing and exploring the bounded search tree for solving the vertex cover problem (The vertex cover problem is explained in chapter 2) of size bounded above by k.

The main idea of method 1 is to choose any edge uv and recursively branch into two cases: either we take u into the vertex cover, or we take v. In fact, the root of the search tree consists of an empty vertex cover and a graph G. we pick any edge uv $\in$ E. We construct the children of the root corresponding to the following 2 choices: In the left node we consider u to be in the vertex cover and in the right node we consider v to be in the vertex cover. So in the first case we delete u from the graph together with all its incidents edges and in the second case we do the same but this time with v deleted from the graph. There is no need to search the tree beyond height k since the size of the vertex cover should be of size at most k. This algorithm runs in O (n * 2^k).

Method 2 consists of choosing some vertex and recursively branch into two cases: either we take this vertex into the vertex cover, or we take all its neighbors. In fact, we start with an empty set of vertex cover and an undirected graph G. Then we pick any vertex u $\in$ V. In any vertex cover, either u or all its neighbors should be in the vertex cover to cover the edges incidents from u. So we create children of the root with 2 possibilities: The left node has {u} and G-u, and the right node has N (u) and G-N (u). In other words, in the case of the left child, we have determined that u may be in our vertex cover and so it must be deleted from G together with all its incidents edges, as they are all now covered by the vertex {u}. And in the case of the right node, we have determined that u is not in our vertex cover and so we must take all its neighbors, the r vertices, in the vertex cover and delete them from G together with all its incidents edges, as they are all now

covered by the vertices N (u) and so on (figure 4.2). At each level in the search tree the size of the vertex cover will increase by one and hence k will be decremented by at least one. In Thus, if we create a node at height at most k in the tree that results in a subgraph with no edges, then a vertex cover of size at most k has been reached. In this case, the size of the search tree is smaller than 2^k. Figure 4.3 represents the bounded search tree algorithm of this method.

The upperbound on the bounded search tree of the vertex cover problem can be improved further. Table 3.1 represents the progress done in this respect. But what about its practicality?

The bounded search tree technique is a technique that can often be refined by considering even more complicated cases. However, this allows for repeated improvements in theoretical performance but not in the practical performance. In fact, most of the programs that implement those best-known algorithms skip some cases as they are very complicated and whose overhead could sometimes lead to slow implementations.

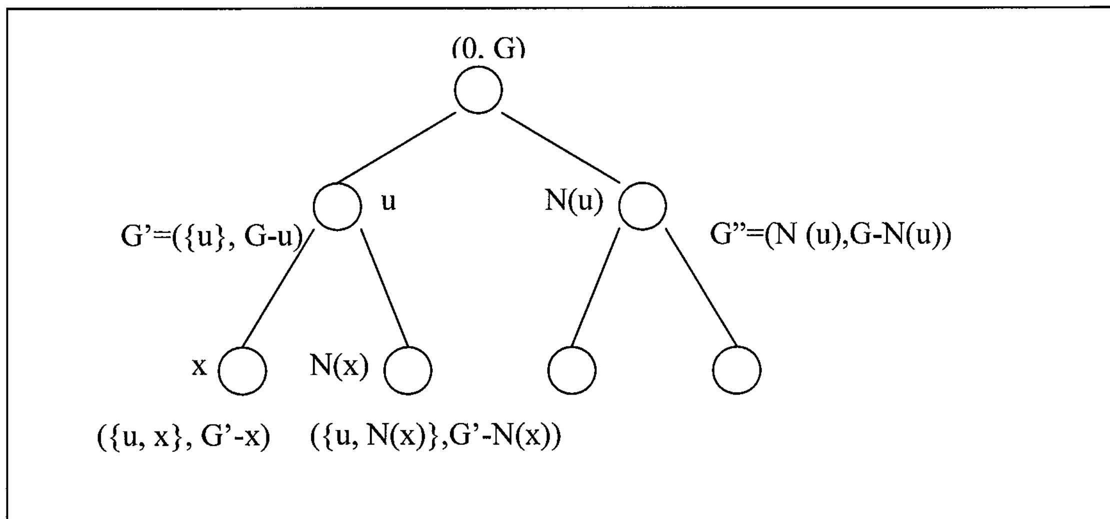


Figure 4.2 - The bounded search tree of the Vertex Cover algorithm

VC-BST (Graph G=(V,E), integer k)

Input: A graph G = (V; E) and an integer k (n = |V |; m = |E|).

If G has no edges then
    Return *true*;

If k = 0 then
    Return *false*;

Select the vertex v of highest degree {v, N (v)} ∈ E
Compute G' = G [V – v]
Compute G" = G [V – N (v)]
Return VC-BST (G', k – 1) or VC-BST (G", k – N (v))

Figure 4.3 - The Pseudo-Code of the bounded search tree of the Vertex Cover algorithm

Table 4.1 – The upperbounds of different algorithms for solving vertex cover problem

Year	Author(s)	Upperbound
1993	Buss	$k \cdot n + 2^k \cdot k^{2k+2}$
1998	Balasubramanian, Fellows and Raman	$k \cdot n + 1.324718^k \cdot k^2$
1999	Downey, Fellows and Stege	$k \cdot n + 1.31951^k \cdot k^2$
1999	Niedermeier and Rossmanith	$k \cdot n + 1.29175^k \cdot k^2$
1999	Stege and Fellows	$k \cdot n + \max\{1.25542^k \cdot k^2, 1.2906^k \cdot k\}$
2000	Niedermeier and Rossmanith	$k \cdot n + 1.2906^k$
2001	Chen, Kanj and Jia	$k \cdot n + 1.2852^k$

4.3 Interleaving

Most of the FPT problems are solved by using kernelization as a first stage and branching as a second stage. However, it is possible to combine kernelization and branching by re-applying kernelization rules at every node of the search tree. In 1999, Niedermeier and Rossmanith called this method *interleaving*. They showed that interleaving speeds up fixed-parameter algorithms. In what follows, we consider the notations used in [14] to illustrate the utility of interleaving.

Let (X, k) be an instance of an FPT problem that can be solved by kernelization and branching. Let us assume that Kernelization takes $p(|X|)$ steps and the resulting instance is of size $q(k)$ where p and q are polynomial functions. On the other hand, the expansion of a node in the search tree takes $r(|X|)$ steps and the resulting search tree is of size $O(\xi^k)$. The total time of the overall algorithm is equal to $O(p(|X|) + r(q(k)) \xi^k)$. By applying interleaving, the overall time becomes $O(p(|X|) + r(\xi^k)$. From this, we can deduce that the improvement that interleaving does is to get rid of the factor q(k) which is replaced by a small constant.

In fact, the main idea developed in [14] is simply the re-application of kernelization at every node of the search tree. If kernelization applies, we obtain a reduced instance which is then used as input to expand the search tree. A closer look shows that it takes more time to expand a node in the search tree by doing re-kernelization at each level of the

search tree. However, these additional reductions to problem kernel may result in a decrease of the instance size. Since the splitting of the search tree depends heavily on the size of the input, these reductions decrease the time to expand the search tree.

To illustrate further, take for instance the vertex cover problem. The fastest algorithm to solve the vertex cover problem is that of Chen, Kanj and Jia which is $O(1.2852^k * k + k*n)$. Due to the interleaving technique, it is now possible to remove the polynomial factor in the exponential term of the total complexity time in this case k. Thus, the overall time complexity of Chen, Kanj and Jia algorithm is now $O(1.2852^k + k*n)$.

Some authors find interleaving a real enhancement for FPT algorithms as in [4]. Others find it "a technique whose impact can sometimes be dramatic, especially on small or sparse instances"[15]. In addition, in [15] it is also noted that even for large, dense or highly regular graphs, the improvement that the interleaving does is negligible.

4.4 Pseudo-kernelization

Pseudo-kernelization was first introduced by Dr. Abu Khzam in 2003. It is a new technique that improves branching algorithms [2]. In general and as mentioned before, the main focus of bounded search tree techniques is to minimize as much as possible the maximum number of children of a node and to limit the size of the search tree. Pseudo-kernelization works well in realizing this aim.

Let P be a FPT problem with input (X, k). We assume that P can be solved using both kernelization and bounded search tree technique. Applying pseudo-kernelization results either in a better worst-case run time or better reduction to the size of the input instance. In fact, this new technique seeks to improve the worst-case run time for the branching algorithm if a branching condition C(P) is satisfied and gives better reduction to the size of the input instance if the same branching condition C(P) is not satisfied.

A branching condition C(P) is a constraint that could be applied on the input instance. If this condition returns a "true" answer, then it is applied on the input instance to reduce the size of the search tree. If this condition returns a "false" answer, a pseudo-reduction rule is used to reduce the size of the (current) input instance.

When comparing interleaving with pseudo-kernelization, we find out that the main difference lies in that pseudo-kernelization always looks for the changes of the input instance appearing during the branching phase and tries to apply further reduction by applying different reduction rules while interleaving applies the same reduction rule, applied in the preprocessing phase, during the interleaving phase.

The pseudo-kernelization of 3-Hitting Set will be described in detail in the following chapter. In addition, we will discuss the pseudo-kernelization technique when applied to the general d-Hitting Set in chapter 6.

Chapter 5

3-Hitting Set: Recent Bounded Search Tree Algorithms and Pseudo-Kernelization

The general Hitting Set problem is defined as follows: Given a collection C of subsets of a finite set S and a positive integer k. The problem is to find a subset of S whose size $\leq$ S, such that this subset contains at least one element from each subset in C. The Hitting Set problem for size three is called 3-Hitting Set. Figure 5.1 represents an example of the 3-Hitting Set problem. In 3-Hitting set, the collection C of subsets is of size three. When every subset in C is of size not exceeding a fixed positive integer d, the problem is called d-Hitting Set. Our focus in this chapter is on the 3-Hitting Set problem.

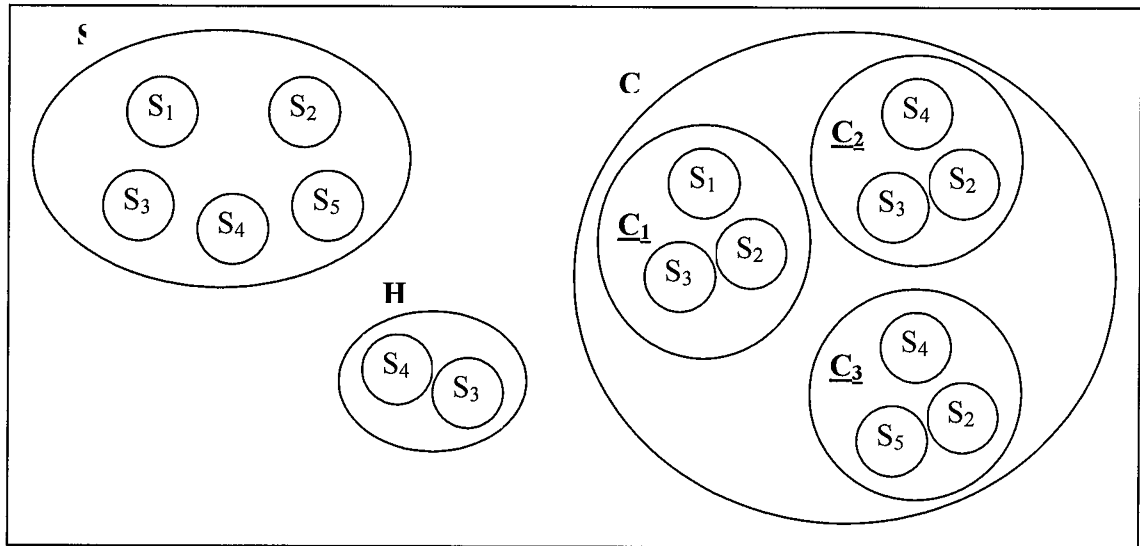


Figure 5.1 - 3-Hitting-Set Example

3-Hitting Set is NP-Complete [16]. However, it is fixed parameter tractable and can be solved using a search tree in a time bounded above by $O(2.5616^k + n)$ [17]. In this chapter, we describe two recent 3-Hitting Set algorithms. The first one is due to Niedermeier and Rossmanith and runs in $O(2.270^k + n)$ time. The other one is that of Henning Fernau, which runs in $O(2.1788^k + n)$. In addition, we discuss the kernelization and pseudo-kernelization techniques.

In the first section, we describe a kernelization technique. In section 5.2 we explain in details the two bounded search tree techniques. In the third section we discuss the effect of interleaving. In the fourth section, we discuss the bounded search tree of Niedermeier and Rossmanith compared to that of Fernau and finally in the last section we describe some pseudo-kernelization techniques.

5.1 Kernelization of 3-Hitting Set

Kernelization of 3-Hitting Set consists of reducing the original instance to a kernel of size $O(k^3)$ elements. The idea is based mainly on two rules:

Rule 1: If two elements x and y appear simultaneously in more than k size-three subsets in the collection C, then all these subsets can be deleted from C and replaced with one size-two subset containing only x and y.

Rule 1 can be proved as follows. Since each size-three subset appears only once in the collection C and since there are more than k size-three subsets containing x and y, this means that there are more than k elements other than x and y in the corresponding sets. If neither x nor y are in the hitting set, than these more than k size-three subsets are covered by these "third" elements. But since the size of the Hitting set |H| is k, this implies that we have to include at least x or y to be in the hitting set H.

Rule 2: If an element x appears in more than k^2 size-three subsets in the collection C, then x should be in the hitting set H and hence all size three subsets containing x can be deleted from C.

Rule 2 can be proved as follows. After applying rule 1, we know that x occurs along with another element say y at most k times. If x is not in the hitting set H, then we need at least (k + 1) elements to hit the more than k^2 size three subsets. This is impossible since the size of the hitting set is at most k. Hence, x should be in the hitting set H.

After applying rule 2, we know that each element x appears at most in k^2 size-three subsets in the collection C (since if x appears more than k^2 times, x should be in the Hitting set H). Since the size of the hitting set is at most k elements, C can consist of at most $k * k^2 = k^3$ elements. By this, we reduce the input instance to a kernel instance whose size is a polynomial function of the parameter k.

5.2 Bounded Search Tree Algorithms of the 3Hitting-Set Problem

5.2.1 Bounded Search Tree Algorithm by Niedermeier and Rossmanith

The structure of the search tree algorithm used in the paper of Niedermeier and Rossmanith is as follows: Deal with the simple cases, subsets of size two, elements of degree three, elements of degree at least four and finally the collection of subsets that is 2-regular. Simple cases consider subset of size one, element of degree one and dominated elements. Note that the sequence of these steps is important. To estimate the size of the search tree, two notations are considered: B_k and T_k. When C contains size two-subsets, the size of the search tree is at most B_k. On the other hand, when all subsets in C are size-three subsets, the size of the search tree is at most T_k. By analyzing the time complexity of this algorithm, we get $O(2.270^k + n)$.

The bounded search tree technique of Niedermeier and Rossmanith can be evaluated as follows:

1- Complicated: It is complicated since it incorporates many cases.
2- Tedious Practicality: Trying to implement this technique is a very tedious task. When excluding these subcases the code runs faster [3].
3- Unverifiable: Since the search tree algorithm is too intricate, it is hard to prove its correctness.

5.2.2 Bounded Search Tree Algorithm by Henning Fernau

5.2.2.1 General Notions and Definitions

Let G be a hypergraph such that G = (V,E) where V is a finite set of vertices and E is a set of hyper-edges. The annotations used by Henning Fernau [3] in the analysis of the Bounded Search Tree are as follows:

$\delta(e)$ = The cardinality of a hyperedge e also called degree of a hyperedge.
$\delta(v)$ = The cardinality of a set of edges that contains a vertex v also called degree of v.
$\#^d S$ = The number of edges or vertices of the edge set or vertex set S which have degree d.

$\delta^d(v)$ = The number of hyperedges of degree d that contains the vertex v.

$T^{\ell}_d(k)$ = The number of leaves in the search tree that has ℓ edges (with parameter k) that have a degree at most d-1.

5.2.2.2 The Algorithm of Fernau

Henning Fernau introduced a new bounded search tree algorithm. This algorithm is based on: The reduction rules, simple instances and heuristic priorities.

Reduction rules:

Reduction rules are used to reduce the input instance into smaller instance. This results in a modification of not only the input instance but also the parameter k. these reduction rules should be applied at each node of the search tree. These reduction rules are as follows:

1. Hyper-edge domination: If a subset f is included in another subset e, i.e. if $f \subset e$ then e is deleted.
2. Tiny edges: Delete subsets of size one and put the corresponding elements into the hitting set.
3. Vertex domination: An element x is dominated by another element y, if x does not occur in a subset without having y occurring in the same subset. Since x is dominated by y and y may occur in other subsets, the co-occurrence of x and y could be replaced by y. Thus, a subset of size three containing x and y will be replaced by a subset of size two and a subset of size two containing x and y will be replaced by a subset of size one.

After applying, the reduction rules, it is guarantee that the reduced graph G has no vertex of degree less than two. All vertices have degree at least 2.

Simple instances:

Simple cases are instances that can be solved efficiently. In other words, these are the instances for which polynomial time algorithms exists. They act as the "stopping situations" in the algorithm. In the case of hitting set problem, instances that have maximum vertex degree of two are defined as simple instances.

Heuristic priorities:

Tells us which subset and which element the algorithm will select for branching. For the 3-Hitting set problem, the heuristic priorities are as follows:

1. Branch by considering subsets of size 2
2. Among all subset of size 2, choose a subset of size 2 whose elements occur the most in subsets of size 3.
3. Choose the elements that have the highest degree

The running time of the algorithm is $O(2.1788^k + n)$. It is analyzed by considering an auxiliary parameter l. l is the number of subsets that are of size 2. By considering an instance with one subset of size two and an instance with two subsets of size two, we reach a run time complexity of $O(2.2470^k + n)$.

Table 5.1 represents the different cases considered when the graph has only one subset of size 2. The resulting upperbound for one subset of size two is:

$$T^1{}_3(k) = T^0{}_3(k-1) + T^2{}_3(k-1)$$

Table 5.1 - The different cases considered when there is one subset of size two

	Cases of one subset of size 2	Upperbound of T^1_3
1	{x,y} ; $\delta(x) \geq \delta(y)$; $\delta(y) = 2$	$T^2_3(k-1)$
2	{x,y} ; $\delta(x) \geq \delta(y)$; $\delta(y) > 2$ The worst case is $\delta(x)=\delta(y) = 3$	$T^0_3(k-1) + T^2_3(k-1)$

Table 5.2 represents the different cases considered when the graph has only two subsets of size 2. Two main cases are considered: when the 2 subsets of size two are disjoint and when the two subsets of size two are not disjoint.

Table 5.2 - The different cases considered when there are two subsets of size two

	Cases of two subsets of size 2	Upperbound T^2_3
1	{x,y} and {u,v} 2 subsets of size 2 that are disjoint	$T^1_3(k-1) + T^3_3(k-1)$
2	{x,y} and {y,z} 2 subsets of size 2 that are not disjoint $\delta(y) \geq 2$ and $\delta(z) \geq 2$ $\delta(x) = 2$	$T^0_3(k-2) + T^1_3(k-1)$
3	{x,y} and {y,z} 2 subsets of size 2 that are not disjoint $\delta(y) \geq 2$ and $\delta(z) \geq 2$ $\delta(x) = 3$ and $\delta(y) = 2$	$T^1_3(k-1) + T^1_3(k-2)$
4	{x,y} and {y,z} 2 subsets of size 2 that are not disjoint $\delta(y) \geq 2$ and $\delta(z) \geq 2$ $\delta(x) \geq 3$ and $\delta(y) \geq 3$	$T^1_3(k-1) + T^2_3(k-1)$
5	{x,y} and {y,z} 2 subsets of size 2 that are not disjoint $\delta(y) \geq 2$ and $\delta(z) \geq 2$ $\delta(x) \geq 4$	$T^0_3(k-1) + T^2_3(k-2)$

Solving the following equations:

$$T^0_3(k) = T^0_3(k-1) + T^2_3(k)$$
$$T^1_3(k) = T^0_3(k-1) + T^2_3(k-1)$$
$$T^2_3(k) = T^1_3(k-1) + T^2_3(k-1)$$
$$T^2_3(k) = T^0_3(k-2) + T^1_3(k-1)$$
$$T^2_3(k) = T^0_3(k-1) + T^2_3(k-2)$$

We reach a run time complexity of $O(2.2470^k + n)$.

By analyzing instances with different value for the parameter l and by considering more reduction rule, different heuristic priorities and kernelization, Fernau reached a run time complexity of $O(2.1788^k + n)$.

5.3 Interleaving of the 3Hitting-Set Problem

Recall that interleaving is simply applying kernelization at each node of the search tree. When applying interleaving technique to the bounded search tree of Niedermeier and Rossmanith, the time complexity is reduced from $O(2.270^k k^3 + n)$ to $O(2.270^k + n)$, replacing k^3 by a small constant.

However, from the practicality point of view, interleaving effects:

1. Vary greatly depending on the input size: applying interleaving on small instances has unsatisfactory effect and applying it to large instances gives negligible improvements [16].
2. Result in many overheads: when applying kernelization at each node of the search tree, we must keep track of the changes that may result [16].

5.4 Comparison between the Algorithm of Niedermeier and Rossmanith and that of Fernau

In the following table (Table 5.3), the algorithm of the bounded search tree of Niedermeier and Rossmanith is compared to that of Henning Fernau.

Table 5.3 – Comparison between the Algorithm of Niedermeier and Rossmanith and that of Fernau

Niedermeier and Rossmanith Algorithm	Henning Fernau Algorithm
Uses Kernelization	Does not use kernelization
Complicated	Simple and modular
Intricate analysis	Generic analysis
Run time of $O(2.270^k + n)$	Run time of $O(2.1788^k + n)$
Use one parameter which is the parameter k	Uses two parameters: The parameter k and an auxiliary parameter l which is the number of subset of size two

5.5 Pseudo-Kernelization of the 3Hitting-Set Problem

5.5.1 Quadratic Pseudo-Kernel of the 3Hitting-Set Problem

The Pseudo-kernelization of the 3Hitting-Set problem introduced by F. Abu-Khzam aims at either reducing the problem kernel to a size bounded above by $4k^2$ or reducing the bounded search tree run time to $O(2.056^k + n)$.

The pseudo-kernelization technique considers first a condition. The condition considered for 3Hitting-Set is as follows:

C_t: In a collection C, there exist two elements x and y $\in$ S that co-occur in at least t subsets. In other words, $co\{x,y\} \geq t$.

If C_t holds:

If C_t holds, the run time complexity of the 3Hitting-Set problem will be reduced to $O(2.056^k + n)$ for $t = 5$.

In fact, if C_t holds, one branch is that x is put in the hitting set H, another branch is that y is put in the hitting set H and a final branch is that if neither x nor y are put in the Hitting set H then t "third" elements are put in the hitting set H. so this gives $T(k) = 2T(k-1) + T(k-t)$. For t=5 solving T(k) yields to a branching factor of 2.056 i.e. $O(2.056^k + n)$

If C_t does not hold:

If C_t does not hold, a pseudo-reduction rule is applied to reduce the input instance. In particular, If C_5 does not hold, the input size is reduced to $O(4k^2)$. In the absence of C_5, the pseudo kernel rule is as follows: if an element $x \in S$ appears in more than 4k subsets of the collection C, then x should be in the hitting set H.

In fact, each element x may occur along with another element at most 4 times. To cover the 4k+ 1 subsets in the collection C, we need k elements from the hitting set H. By the pigeon hole principle, there exist one element $\in$ S and that exists with another elements in 5 subsets. This contradicts our assumption that each element co-occurs with another element at most in four subsets. Hence, x should be in the hitting set H. As a result, x appears at most in 4k size-three subsets of C (since if x appears in more than 4k subsets, x should be in the Hitting set H). Since the size of the target hitting set (if it exists) is at most k, C can consist of at most $k * 4k = 4k^2$ elements.

Chapter 6

Pseudo-Kernelization and d-Hitting Set

We introduce, develop and analyze the pseudo-kernelization technique for d-Hitting Set. Applying this technique improves on existing fixed-parameter algorithms.

In what follows, we will use co(x,y) to denote the number of the subsets in C that contains x and y simultaneously. In addition, we will use T(k) to denote the number of leaves in a search tree.

In the work of Niedermeier and Rossmanith, a general algorithm for d-Hitting set is presented. This algorithm has a running time of $O(\alpha k + n)$ where

$$\alpha = (d-1) + 1/(d-1) + O(d^{-3})$$

While section 6.1 elaborates on this algorithm, section 6.2 analyzes and presents the pseudo-kernelization for the d-Hitting Set problem.

6.1 Bounded Search Tree for the d-Hitting Set Problem

The algorithm introduced in [4] for d-hitting set problem works first by removing all dominating elements. After that, a subset $s = \{x_1, x_2, \ldots, x_d\}$ is selected, we can branch by choosing:

1- x_1 is in the Hitting set: Let T(k) be the number of leaves in the search tree, then in this case we have at most T(k-1) leaves.

2- x_1 is not in the hitting set and but x_i for $i = 2,3,\ldots,d$ is in the Hitting set: Let B(k) be the number of leaves in the search tree where there is at least one set of size $\leq$ d-1. For each branch x_i, there is a subset s' in C such that $x_1 \in s'$ and x_i

$\notin$ s'. Hence, after excluding x_1 from the set s' and including x_i in the hitting set the subset s' will be of size at most d-1.

Hence, the upperbound on the size of the search tree is: T(k) = T(k-1) + (d-1) B(k-1). Similarly, if the subset is of size at most d-1, we get an upperbound : T(k) = T(k-1) + (d-2) B(k-1). By solving this recursion we get a running tile algorithm of $O(\alpha k + n)$ where

$$\alpha = (d-1) + 1/(d-1) + O(d^{-3})$$

Table 6.1 shows the running time of the algorithm described above for several values of d as presented in [4].

Table 6.1 - Results of T(k) of the algorithm of Niedermeier and Rossmanith for solving d-hitting set problem

d	3	4	5	6	7	9	10	20	50	100
T(k)	2.41	3.30	4.23	5.19	6.16	8.12	9.11	19.05	49.02	99.01

6.2 Pseudo-Kernelization for the d-Hitting Set problem

Recall that the Pseudo-Kernelization technique considers a favorable branching condition whose fulfillment leads to a better running time of the search tree and whose absence leads to a better reduction of the input instance.

Consider the following branching condition:

C_t (d-hitting Set): $\{x_1,x_2,\ldots x_{d-1}\} \in S$ and $co(x_1,x_2,\ldots,x_{d-1}) \geq t$

<u>C_t holds</u>

Let $E(x_1,x_2,\ldots,x_{d-1}) = \{x_d \in S$ such that $(x_1,x_2,\ldots,x_{d-1},x_d)$ is a subset in C$\}$. Since every subset in C occurs only once, if C_t holds then we will have the size of $E(x_1,x_2,\ldots,x_{d-1}) \geq t$. Then the branch will be according to the following possibilities:

- x_1 is in the hitting set: In this case, we will have T(k-1) leaves
- $x_j \notin H$ and $x_i \in H$ for $j < i$; In this case, we will have (d -2)T(k-1) leaves.
- $E(x_1,x_2,\ldots,x_{d-1})$ is in the hitting set: this results in T(k-t) leaves.

All in all, we will get T(k) = T(k-1) + (d-2) T(k-1) + T(k-t) = (d - 1) T(k-1) + T(k-t). Solving this equation, we will get $x^k = (d-1)\, x^{k-1} + x^{k-t} \Rightarrow x^t = (d-1)\, x^{t-1} + 1$.

Table 6.2 represents the run time T(k) for different value of d and for t = 5.

Table 6.2 – Results of T(k) of the Pseudo-kernelization algorithm for solving d-hitting set problem

d	3	4	5	6	7	9	10	20	50	100
T(k)	2.055	3.012	4.003	5.001	6.000	8.000	9.000	19.000	49.000	99.000

As a result, referring to table 6.2, we notice that the fulfillment of the condition Ct gives a better running time than that of Niedermeier and Rossmanith presented in table 6.1.

C_t does not hold

Since C_t does not hold, we will have $\{x_1,x_2,\ldots x_{d-1}\} \in S$ and $co(x_1,x_2,\ldots,x_{d-1}) < t$. Now, we consider the following reduction rules:

C^{d-2}_t: If $co(x_1,x_2,\ldots x_{d-2})$ appears in more than $(t-1) * k$ subsets in C, then either x_1 or $x_2, \ldots, x_{d-2}$ should be in the Hitting set.

In fact, if neither x_1 nor $x_2, \ldots,$ nor x_{d-2} are in the hitting set, then there are at most k elements in the hitting set that cover at least $((t-1) * k) + 1$ subsets, so there is at least one element that co-occurs with $(x_1,x_2,\ldots x_{d-2})$ in t subsets. This contradicts the absence of C_t which is $co(x_1,x_2,\ldots,x_{d-1}) < t$. Hence, either x_1 or $x_2, \ldots, x_{d-2}$ should be in the Hitting set.

This condition is applied until there are no more $co(x_1,x_2,\ldots x_{d-2})$ appearing in more than $(t - 1) * k$. Hence, $co(x_1,x_2,\ldots x_{d-2}) \leq (t - 1) * k$.

$\mathbf{C^{d-3}}_t$: If $co(x_1,x_2,\ldots x_{d-3})$ appears in more than $(t-1) * k^2$ subsets in C, then either x_1 or x_2, ..., x_{d-3} should be in the Hitting set.

The proof is the same as that of C^1_t. That is if neither x_1 nor x_2, ..., nor x_{d-3} are in the hitting set, then there are at most k elements in the hitting set that cover at least $((t - 1) * k^2) + 1$ subsets. So there is at least one element that co-occurs with $(x_1,x_2,\ldots x_{d-3})$ in $(t - 1) * k + 1$ subsets. This contradicts the absence of C^1_t.

We apply C^{d-4}_t, $C^{d-5}_{t,\ldots,}$ C^2_t to the given instance in the same way as C^{d-2}_t and C^{d-3}_t until pseudo-reduction rule is applied as follows:

Rule 1: If an element $x \in S$ appears in more than $(t - 1) k^{d-2}$ subsets in C, then x should be in the Hitting set.

In fact, if x is not in the hitting set, then there are at most k elements in the hitting set that cover at least $((t - 1) k^{d-2}) + 1$ subsets, so there is at least one element that co-occurs with x in $(t - 1) k^{d-3} + 1$ subsets. This contradicts the absence of C^2_t which is $co(x_1,x_2) \leq (t - 1) k^{d-3} + 1$. Hence, x should be in the Hitting set.

Since k elements should cover all subsets in C and none of these k elements appear in more than $(t - 1) k^{d-2}$, the collection C has at most $k * (t - 1) k^{d-2} = (t-1) * k^{d-1}$.

Rule 2: If the collection C has more than $(t-1) * k^{d-1}$ subsets, then no solution of size k can hit all the subsets of C. Hence, the input instance is a No instance.

Improving the algorithm of Henning Fernau described in section 6.2.2, the algorithm of the pseudo-kernelization can be described as follows:

1- If C_t holds, the algorithm branches using the pseudo-kernelization branching.

2- If C_t does not hold, it checks for the pseudo-reduction rule.

3- If the solution is not reached, we check the number of subsets in C to know whether there is or there is not a solution.

4- If no solution is found, the algorithm returns No.

5- Otherwise, we select a branching element based on Henning Fernau algorithm and placed it in the hitting set.

The algorithm repeats the above mentioned steps until a solution is found or no solution is found.

Chapter 7

An Improved Algorithm for 3-Hitting Set Problem

In this chapter, we present an algorithm that combines the heuristic of H. Fernau Algorithm described in section 6.2.2 and the branching of pseudo-kernelization algorithm described in section 6.5.

In what follows, we will refer to "Pure Branching" to denote the algorithm of H. Fernau for the 3-Hitting Set problem and "Hybrid Algorithm" to denote our improved algorithm for the 3-Hitting Set problem.

7.1 The Hybrid Approach

While pure branching seeks to find the highest degree element, pseudo-kernelization seeks to find highly co-occurring pairs. The former algorithm takes at each level of the search tree one element x of highest degree and branch according to whether x is in the Hitting set or not. The latter algorithm takes at each level of the search tree two elements x and y that have highest co-occurrence and branch according to whether x or y or all the "third elements" that occurs with x and y are in the Hitting Set. By combining the "highest degree element" and "highest co-occurrence" concepts of H. Fernau and F. Abu Khzam respectively, we have developed our new algorithm.

The hybrid algorithm checks if all sets of the input instance are of size three. If so, it considers first the couples that co-occur the most in the given input instance. Let *comax* be the value of the highest co-occurrence among all couples. Among the different couples that highly co-occur, we choose a couple whose one of its elements has the highest degree. Such a couple is further refined to one couple by choosing among them the

couple (x , y) whose second element has the highest degree. Our new algorithm branches by considering 3 possibilities:

1- x is in the Hitting set.
2- x is deleted from the input instance but not placed in the hitting set and y is in the Hitting set.
3- x and y are deleted from the input instance but not placed in the hitting set and “the third elements” that occur with x and y are placed in the hitting set.

If there are other couples that co-occur in more than *comax* sets of the input instance, the steps mentioned before are repeated until there are no couples that co-occur in more than comax sets of the input instance.

Before considering the next highest co-occurrence in the resulting input instance, we check the pseudo-reduction rule. Each element that is neither deleted from the input instance nor placed in the Hitting set is checked if it occurs in more than comax * k sets of the input instance. In other words, each element is checked if its degree is greater than *comax* * k. if this condition is satisfied, these elements are deleted and placed in the hitting set. In addition, if the number of edges are more than *comax* * k^2, the algorithm return No solution.

If the sets of the resulting instance are not all of size three, we choose an element x for branching following the heuristic priorities of Fernau as follows:

1- x is in the hitting set
2- x is deleted from the instance but not placed in the hitting set

The pseudo-code of our new algorithm is presented in chapter 8.

Chapter 8

Experimental Analysis

In this chapter, we describe the implementation of the pure branching and the hybrid algorithm. Moreover, we conduct experiments on randomly generated instances, in order to show the utility of our approach. For this purpose, we developed a random generator for 3-Hitting Set instances of pre-determined edge densities.

Programs were developed using C++ and run on a Pentium III machine. A detailed description of the data structure and functions used in the implementation of the above mentioned techniques will be presented in appendix I.

8.1 Implementation Details

Our implementation is mainly divided into 4 modules:

1- Reduction rules module: This code detects the reduction rules described in [3] and removes the elements that satisfy any of the reduction rule from the input instance. The pseudo-code of this part is shown in figure 8.2.

2- Priorities module: This module chooses an element and places it in the hitting set by following the heuristics rules described in [3]. The pseudo-code is shown in figure 8.3.

3- Check-Pseudo-Kernelization branch module: This module considers the condition described in Chapter 4. The pseudo-code is presented in figure 8.5. It contains two modules:

   a. Pseudo-Kernel branch module: This module checks if the condition described in [2] applies. It gives the possible branching elements that should be in the hitting set. This is done by choosing the highly co-

occurring couple that contains at least one element of highest degree. The pseudo-code is shown in figure 8.6.

b. Pseudo-Kernel reduction module: This module acts when the condition described in [2] does not apply. So it checks if there are elements that satisfy the reduction rule described in [2] and places these elements in the hitting set. The pseudo-code of is shown in figure 8.7.

The pseudo-codes of the pure branching and the hybrid algorithm are shown in figures 8.1 and 8.4 respectively.

8.2 Random Generator

We developed a program in C++ that creates files that are used as input instances for our algorithms. The input of the random generator is as follows:

1- The name of the output file
2- n : The number of elements in S
3- p : The probability for generating m, which is the number of subsets of size three
4- k : The size of the solution

And the output is a file that includes the following:

1- m : The size of the subsets
2- n : The number of elements in S
3- k : The size of the solution
4- The subsets of size three, each on a separate line.

The pseudo-code of our random generator is described in figure 8.8.

8.3 Statistics and Conclusion

Table 8.1 represents the results derived when running the pure branching and the hybrid algorithm. These experimental results reveal the following:

1- The hybrid algorithm always runs faster than the pure branching.
2- The element of the couple that highly occurs in the input instance and has the highest degree is more likely to be in the hitting set.
3- There are some instances where the reduction rule of pseudo-kernelization applies and hence the algorithm places elements directly in the hitting set without considering them as nodes in the bounded search tree. This eventually makes the hybrid algorithm runs faster.

Input

C : A collection of subsets of size 3
m : the size of C
n : The number of elements in S
k : The size of the solution.

Output

No : if No solution of size k exists. Return No.
Yes: If there is a solution of size at most k, returns Yes and generates the solution.

```
Branching(C , m , n , k)

If ( m = 0 ) and ( k ≥ 0 )
        Branch_flag = "Yes";
        Generates the solution;
Else
If ( m > 0 ) and ( k ≤ 0 )
        Branch_flag = "No";
Else
Take a backup of C, m, n and k
x = Heuristic_priorities(C, m , n , k)
Place x in the Hitting Set
k = k -1
Reduction_rules(C, m, n, k)
If ( Branching (C, m , n , k) = "No")
        Restore C, m, n and k
        Delete x from the input instance without placing it in the hitting set
        Reduction_rules(C, m , n , k)
        Branch_flag = Branching(C, m , n , k)

Return Branch_flag;
```

Figure 8.1 - The pseudo-code of Fernau Branching Technique.

Input

C : A collection of subsets of size 3
m : the size of C
n : The number of elements in S
k : The size of the solution.

Output

Return the input instance after no reduction rules are applied.

```
Reduction_rule(C , m , n , k)

Repeat

        If  (edge domination = "Yes")
                Delete the edges that are dominated by other edge
        If  (subset of size one = "Yes")
                Put the element of the subset of size one in the hitting set
                k = k - 1
        If  (vertex domination = "Yes")
                Delete the elements that are dominated by another element

Until (edge domination = "No") and (subset of size one = "No") and (vertex
      domination = "No")
```

Figure 8.2 - The pseudo-code of "Reduction_rules".

```
Input

C : A collection of subsets of size 3
m : the size of C
n : The number of elements in S
k : The size of the solution.

Output

Return an element x ∈ S.

Heuristics_Priorities(C , m , n , k)

If there are subsets of size two
        Max = 0
        For each element x of the subset of size 2
                Max_x = the number of subsets of size three where x occurs –
                the number of subsets of size three where x occurs
                if Max_x > Max
                        Max = Max_x

Else
        Max = 0
        For each element x of the subset of size 3
                Max_x = the number of subsets of size three where x occurs
                if Max_x > Max
                        Max = Max_x

Return x;
```

Figure 8.3 - The pseudo-code of "Heuristic_priorities".

Input

C : A collection of subsets of size 3
m : the size of C
n : The number of elements in S
k : The size of the solution.

Output

No : if No solution of size k exists, return No.
Yes: If there is a solution of size at most k, return Yes and Print the elements that should be in the hitting set.

Pseudo_Kernelization(C , m , n , k)

```
Edge_flag       = "No"
Reduction_flag = "No"

edge_flag = Check_Pseudo_kernelization(C, m, n, k)

If ( m = 0 ) and ( k ≥ 0 )
        Branch_flag = "Yes";
        Print the solution;
Else
If ( ( m > 0 ) and ( k ≤ 0 )) or (edge_flag = "No")
        Branch_flag = "No";
Else
Take a backup of C, m, n and k
If not all subsets are of size three
        x = Heuristic_priorities(C, m , n , k)
        Place x in the Hitting Set
        k = k -1
        Reduction_rules(C, m, n, k)
```

Figure 8.4 - The pseudo-code of our new improved algorithm (cont)

```
else
        x = PK[0]
        Place x in the Hitting Set
        k = k -1
        Reduction_rules(C, m, n , k)
        edge_flag = Check_Pseudo_kernelization
        If  ( Branching (C, m , n , k)= No)
                Restore C, m, n and k
                If not all subsets are of size three
                        Delete x from the input instance without placing it in the
                         hitting set
                        Reduction_rules(C, m, n , k)
                        Branch_flag = Branching(C, m, n , k)
                else
                        Delete x from the input instance without placing it in the
                        hitting set
                        Reduction_rules(C, m, n , k)
                        y = PK[1]
                        Place y in the Hitting Set
                        k = k -1
                        Reduction_rules(C, m, n , k)
                        Check_Pseudo_kernelization(C, m, n, k)
                        If  ( Branching (C, m , n , k)= No)
                                Delete x from the input instance without placing it
                                in the hitting set
                                Reduction_rules(C, m, n , k)
                                Delete y from the input instance without placing it
                                in the hitting set
                                Reduction_rules(C, m, n , k)
                                (For i = 2 to maxco)
                                        z = PK[i]
                                        Place z in the Hitting Set
                                k = k – maxco
                                Reduction_rules(C, m, n , k)
                                edge_flag = Check_Pseudo_kernelization(C, m, n,k)

                        Branch_flag = Branching(C, m, n , k)Else

Return Branch_flag;
```

Figure 8.4 - The pseudo-code of the Pseudo-Kernelization Technique.

Input

C : A collection of subsets of size 3
m : the size of C
n : The number of elements in S
k : The size of the solution.

Output

edge_flag : "Yes" or "No"
If the pseudo-reduction rule is applied, a new instance is returned which is the input instance resulting by deleting all elements that occurs in more than (t – 1) * k sets.

Check_Pseudo_Kernelization(C , m , n , k)

```
maxco = next_maxco

If all subsets are of size three
        next_maxco = get_maxco(C,m)
        If (maxco = 0)
                maxco = next_maxco
        If (maxco <> next_maxco)
                Reduction_flag =  Pseudo-Kernel reduction (C,m,n)
                If (Reduction flag = "No")
                        If  (m > (t -1 ) * k^2)
                                edge_flag = "Yes"
                        else
                                edge_flag = "No"
                else
                        Place all elements whose degree > ( t –1 ) * k in the hitting
                        set
                        K = k- {number of the elements whose degree > ( t -1 ) *k}
                        PK = Pseudo-Kernel branch (C, m, n,next_maxco)
        else
                PK = Pseudo-Kernel branch (C, m, n,maxco)

Return edge_flag;
```

Figure 8.5 - The pseudo-code of "Check_Pseudo_kernelization".

Input

C : A collection of subsets of size 3
m : the size of C
n : The number of elements in S

Output

Reduction_flag : "Yes" or "No"

Pseudo-Kernel reduction (C, m, n)

```
If there exists one element that occurs in more than ( t -1 ) * k sets
        Reduction flag = "Yes"
Else
        Reduction flag =  "No"

Return Reduction_flag;
```

Figure 8.6 - The pseudo-code of "Pseudo-Kernel reduction ".

Input

C : A collection of subsets of size 3
m : the size of C
n : The number of elements in S
k : The size of the solution.
maxco: The value of the maximum co-occurrence in the input instance

Output

PK: an array that contains all the elements on which we have to branch.
maxco: the value of the maximum co-occurrence in the input instance

Pseudo_Kernel branch(C, m, n, k, maxco)

Check the co-occurrence of each couple of the input instance.
maxco = The value of the maximum co-occurrence in the input instance
Between the couples that co-occur in maxco sets choose one couple (x,y) whose one of its element has the highest degree. Let x be the element of (x,y) that has the maximum degree

```
PK[0] = x
PK[1] = y
i = 2;
While i < (maxco + 2)
        PK[i] = the "third element" that occurs with (x,y)


Return PK;
```

Figure 8.7 - The pseudo-code of "Pseudo_Kernel branch ".

Input :
n : The number of elements in S
k : The parameter
p : The probability
the name of the output file

Output
An output file that includes the following:

m : The size of the subsets
n : The number of elements in S
k : The parameter
The subsets of size three

Random_generator (n, k, p, the name of the output file)

```
for(i=0;i<n;i++){
        for(j=i+1;j<n;j++){
                for(k=j+1;k<n;k++){
                        if (rand() % 100 < p) {
                                write on the file  i  j  k
                                compute m = the number of subsets of size three
write on the file m, n, k
```

Figure 8.8 - The pseudo-code of the Random Generator.

Table 7.1- Experimental results of the pure branching and hybrid algorithm.

Graph Name	Number of the subsets of C	Number of the elements in S	Density	k	Run time in s of pure branching	Run time of hybrid algorithm
RHS01	510	40	5	25	**50**	**41**
RHS02	5833	50	30	44	**0.166**	**0.127**
RHS02	5833	50	30	43	**0.173**	**0.134**
RHS02	5833	50	30	42	**17**	**17**
RHS03	3822	50	20	41	**0.202**	**0.126**
RHS03	3822	50	20	40	**34.82**	**31.716**
RHS04	10250	60	30	51	**53.668**	**52.554**
RHS04	10250	60	30	52	**9.489**	**1.484**
RHS05	6738	60	20	50	**128.765**	**122.485**
RHS06	16443	70	30	61	**153.534**	**145**
RHS06	16443	70	30	62	**3.328**	**1.6**
RHS07	10841	70	20	61	**1.165**	**0.743**
RHS08	4167	80	5	64	**10.29**	**2.782**
RHS09	24642	80	30	73	**1.669**	**1.33**
RHS10	5868	90	5	74	**0.532**	**0.415**
RHS11	48557	100	30	92	**20.651**	**4.525**
RHS12	32153	100	20	91	**4.138**	**3.641**
RHS13	165360	150	30	141	**38**	**34**
RHS14	109958	150	20	140	**118**	**75**
RHS14	109958	150	20	141	**26**	**18**
RHS15	241367	170	30	162	**81**	**53**
RHS16	19173	180	2	155	**16**	**5**
RHS17	65539	200	5	183	**27.893**	**18.923**
RHS18	128264	250	5	233	**85**	**84**
RHS19	443689	300	10	288	**412**	**285**

Chapter 9

Conclusion

In this paper, we considered some recent trends in designing fixed parameter algorithms that are used to find exact solutions to some parameterized problems. These methods are: Kernelization, Bounded search tree, and interleaving. The main objective of fixed parameter algorithms is to solve parameterized problems by reducing the input instance as much as possible, and by achieving the least exponential run time. We considered the 3-Hitting Set problem to test the effectiveness of these methods.

In addition, we considered the "Pseudo-Kernelization" technique which is a new fixed-parameter algorithm introduced by Dr. Abu Khzam. When applying this technique to the 3-Hitting Set problem, it has proved to give a smaller branching factor or a smaller reduced instance, compared to the techniques developed earlier.

We extended the work of Dr. Abu Khzam, and we applied the Pseudo-Kernelization technique to d-hitting set problem. Theoretically, we proved that this technique gives either a best run time or the smallest reduced instance of all the algorithms developed so far.

Also, we developed a new improved algorithm that has proved to give the best results for 3-Hitting Set. It is a hybrid algorithm that combines the heuristic of the algorithm of H. Fernau and the branching strategy used in the Pseudo-Kernelization algorithm.

To prove the effectiveness of our newly developed algorithm from a practical perspective, we have developed programs using C++ to implement, test and compare the branching technique of Fernau (which gives the so-far best known run time) and our

hybrid algorithm. The analysis focused on measuring the time needed to reach the solution. We verified that our hybrid algorithm gives better run times than the mere branching algorithm of Fernau.

References

[1] F. N. Abu-Khzam. "Topics in Graph Algorithms: Structural Results and Algorithmic Techniques, with Applications," PhD thesis, Department of Computer Science, University of Tennessee, 2003.

[2] F. N. Abu-Khzam. "Pseudo-Kernelization: A Branch-then-Reduce Approach for FPT problems," journal of Theory of Computing Systems (TOCS), accepted for publication.

[3] H. Fernau: "A Top-Down Approach to Search-Trees: Improved Algorithmics for 3-Hitting Set," Electronic Colloquium on Computational Complexity (ECCC)(073), 2004.

[4] R. Niedermeier and P. Rossmanith. "An efficient fixed parameter algorithm for 3-hitting set," Journal of Discrete Algorithms, 1:89–102, 2003.

[5] H. Gonzalez-Banos and J. Latombe. "A randomized art-gallery algorithm for sensor placement," in Proc. 17th ACM Symp. on Computational Geometry, 2001.

[6] F. Kuhn, P. Rickenbach, R. Wattenhofer, E. Welzl, A. Zollinger. "Interference in Cellular Networks: The Minimum Membership Set Cover Problem," COCOON 2005: 188-198, 2005.

[7] D.P. Ruchkys and S.W. Song, "A parallel approximation hitting set algorithm for gene expression analysis," 14th Symposium on Computer Architecture and High Performance Computing, 2002.

[8] K. Weihe, "Covering trains by stations or the power of data reduction," Proc. of Algorithms and Experiments (ALEX98), pp.1-8, 1998.

[9] D. Kesdogan, L. Pimenidis: "The hitting set attack on anonymity protocols," in Proc. of 6th Information HidingWorkshop (IH 2004). LNCS, Toronto (2004).

[10] M. Fellows, P. Heggernes, F. Rosamond, C. Sloper, J.A. Telle, "Exact algorithms for finding k disjoint triangles in an arbitrary graph," in: Proc. 30thWorkshop on Graph Theoretic Concepts in Computer Science, Lecture Notes in Computer Science,Vol. 3353, Springer, Berlin 2004, pp.235–244, 2004.

[11] S. Gilmour, M. Dras. "A Two-Pronged Attack on the Dragon of Intractability," in Proc. of the 28th Australasian Computer Science Conference (ACSC-2005), 2005.

[12] C. Sloper. "Techniques in Parameterized Algorithm Design," PhD thesis, Department of Informatics, University of Bergen, Norway, 2005.

[13] R. G. Downey, M. R. Fellows: Parameterized Complexity Springer-Verlag, 1999.

[14] R. Niedermeier and P. Rossmanith. "A general method to speed up fixed-parameter-tractable algorithms," Information Processing Letters, 73:125–129, 2000.

[15] F. N. Abu-Khzam, M. A. Langston, P. Shanbhag, and C. T. Symons, Scalable Parallel Algorithms for FPT Problems, Algorithmica, 2005.

[16] M. Garey, D. Johnson, "Computers and intractability: A Guide to the Theory of NP-Completeness. Freeman, San Francisco, 1979.

[17] R. G. Downey, M. R. Fellows, and U. Stege, "Parameterized complexity: A Frame for systematically confronting computational intractability. In contemporary Trends in Discrete Mathmatics: From DIMACS and DIMATIA to the Future, volume 49 of AMS-DIMACS, pages 49-99. AMS Press, 1999.

Appendix I

Technical Details

The parameters

The global parameters used in the implementation of the pure branching and the hybrid algorithm are as follows:

fp : The input file from which the global parameters should be filled.

set : The input file fp is used to fill the array set. It contains all subsets of size three. It is a read only array.
It is a double dimensional array.
Its size is m*3.

no : The size of the array couple.

couple : It is an array that contains all possible co-occurrence of the elements that are neither in the hitting set nor deleted.
Its size is no * 4
no= (tempn * (tempn-1))/2 note that tempn is the number n that is not placed in the hitting set or deleted.
Column 1 and 2 are the co-occurrence x and y.
Column 3 is the number of the co-occurrence of x and y in the given instance.
Column 4 is the status of the couple
1 it is included in the pair array
2 it is to be included in the pair array

0 not include in the pair array

Degree : It represents the degree of each element in the subsets of size three and the subsets of size two.

It is a two dimensional array.

Its size is n * 2.

Column 1 represents the degree of x in the subsets of size three.

Column 2 is represents the degree of x in the subsets of size two.

HS : The hitting set array that represents whether each element is in the hitting set or deleted without placing it in the hitting set or still active in the given instance.

It is one dimensional array.

Its size is n.

The possible value that HS may contain are:

a = The element is still active

o = The element is deleted but not placed in the hitting set

i = The element is included in the hitting set

n = The element is ***newly*** included in the hitting set

m = The element is ***newly*** deleted but not placed in the hitting set

no_pairs : The size of the array pair.

pair : The pair array represents all the subsets that are of size two now.

It is two dimensional array.

Its size is n_pairs*2.

elements : The element array represents the occurrence of each element with the other elements in a subset of size three.

It is a one dimensional array of size n. each node i of the array points to a 2 dimensional array of size DEG * 2. DEG is the degree of the element i in the subsets of size three.

The array element can be drawn like shown in Figure 1.

PK : PK array is the array that contains the couple that has the highest co-occurrence along with their neighbors in a subset of size three.

It is a one dimensional array.

Its size is maxco + 2

maxco : It is the value of the highest co-occurrence of the given instance.

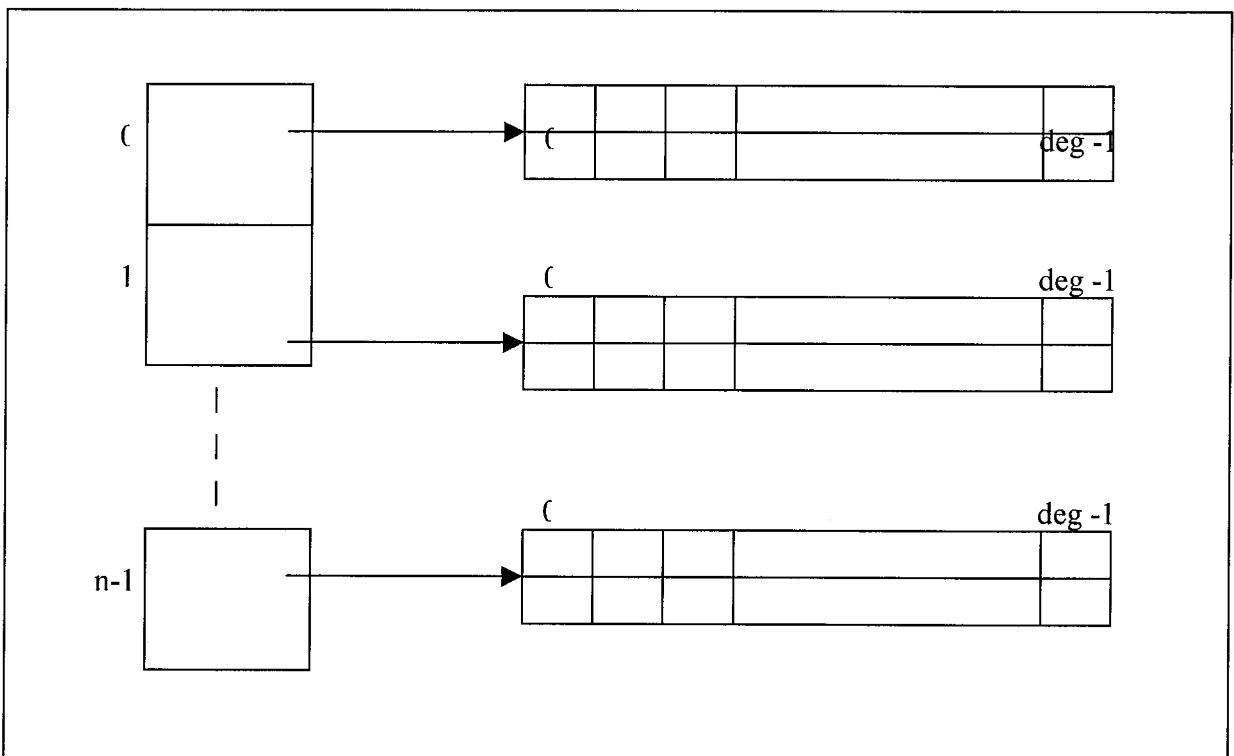


Figure 1- The array element

The Functions Used in the Pure Branching Algorithm

In what follows, we give the input, output and description of each function and procedure used in the implementation of the pure branching algorithm. Figure 2 represents “Main” which is the main procedure of this implementation.

```
Main
k = get_input_file();
init_sets();
fill_sets();
degree = init_degree();
fill_degree(degree);
init_elmnt(degree);
fill_elmnt();
HS = init_HS();
flag = branch(degree,pair,&n_pairs,HS,&k);
if (flag == 1)
        print ("Unfortunately there is no solution\n");
```

Figure 2 - ***Main*** **Procedure**

get_input_file()

Input : The name of the input instance

Output : k

This procedure reads the input file and fills in the value of m n and k and return k.

init_sets()

Input : m and set

Output : set

This procedure initiates the array set of size m * 3.

fill_sets()

Input : m, set and fp

Output : set

This procedure fills the set array from the input file fp.

init_degree()

Input : n and degree

Output : degree

This procedure initiates the array degree of size n * 2.

fill_degree()

Input : m, set and degree

Output : degree

This procedure fills the array degree from the array set.

init_element(degree)

Input : degree, element and n

Output : element

This procedure initiates the array element knowing n and the degree of each element.

fill_element()

Input : element, n, and set

Output : element

This procedure fills the array element from the array set.

init_HS()

Input : n and HS

Output : HS

This procedure initiates the array HS of size n.

get_no(HS,degree)

Input : n, degree and HS

Output : no

This procedure initiates the array HS of size n.

init_couple()

Input : couple and no

Output : couple

This procedure initiates the array couple of size no * 4.

fill_couple(pair,n_pairs,degree,HS)

Input : n, no, element, pair, n_pairs, degree and HS

Output : couple

This procedure fills the array couple. It fills the first 2 columns by reading the HS array and getting each possible co-occurrence of two element of status "a".

```
if ((HS[i]== 'a') and (HS[j]== 'a'))
        couple[u][0] = i;
        couple[u][1] = j;
```

To fill the third and the fourth column, it searches for the co-occurrence of each of the couple in the array couple in the array pair and in the array element. If the couple exists in the array pair the fourth column is set to 1 and it is set to 0 otherwise.

init_pair(n_pairs)

Input : pair and n_pairs

Output : pair

This procedure initiates the array pair of size n_pairs * 2.

fill_pair(pair,n_pairs,degree,HS)

Input : new_pair, no, pair and n_pairs

Output : pair

This procedure fills the array pair. The array new_pair contains the following:

- All the couples of the array pair
- All the couples of the array couple that have a value of 2 in the fourth column.

The array pair is then equal to the array new_pair.

```
Branch(degree, pair,n_pairs, HS, k)
        branch_flag = 0;
        status = check_status(degree);
        if (((status == 1) && (k1 == 0)) ||
           ((status == 1) && (k1 > 0))){
                print("Congratulations you have done\n");
                print the result
                branch_flag = 0 ;
        }
        else
        if (((status = 0) or (status = 2) or (status = 3) or (status = 4) or (status = 5))
           and (k1 = 0) or
           (((status = 1) or (status = 2) or (status = 3) or (status = 4) or (status = 0)
           or (status = 5)) and (k1 < 0)))
                        branch_flag = 1;
        else{
        We take a backup of the following parameters: k, pair, n_pairs, HS and
        degree
        x = get_instance(degree,pair,n_pairs,HS);
        HS[x] = 'n';
        new_pair = set_HS_i(HS,degree,pair,n_pairs);
        pair = new_pair;
        HS[x] = 'i';
        k = k -1;
        new_pair = simple_cases(HS,n_pairs,pair,k,degree);
        pair = new_pair;
```

Figure 3 - ***Branch*** **Procedure (cont)**

```
        if (branch(degree,pair,n_pairs,HS,k)= 1){
                branch_flag = 0;
                We restore the following parameters: k, pair, n_pairs, HS and
        degree
                HS[x] = 'm';
                get_no(HS,degree);
                couple = init_couple();
                fill_couple(pair,n_pairs,degree,HS);
                nocouple2 = adjust_couple_branch(x,degree);
                new_pair = set_HS_o_branch(HS,degree,n_pairs,pair,k,x,nocouple2);
                pair = new_pair;
                new_pair = simple_cases(HS,n_pairs,pair,k,degree);
                pair = new_pair;
                branch_flag =  branch(degree,pair,n_pairs,HS,k);
        }
    }
return branch_flag;
```

Figure 3 - *Branch* Procedure

Branch(degree, pair,n_pairs, HS, k)

Input : m, n, element, no, couple, degree, pair, n_pairs and HS

Output : No if there is no solution of size at most k

Yes if there is a solution of at most k and print the solution

The *Branch* procedure is the main procedure in the implementation of the Branching algorithm of Henning Fernau. The main concept is to select an element x by following the heuristics priorities described earlier in chapter 4 in order to construct the bounded search tree. *get_instance* procedure does the job of selecting an element x. Each time an element x is selected and is to be placed in the hitting set, its status in the array HS is set to “m”. *set_HS_i* procedure deletes all the subsets of size three and of size two of an element whose status in array HS is “m”. After deleting all the subsets of size two and of size three where x occurs, the status of the element x is set to “i”. Then, the reduction rules described in chapter 4 are applied to the resulting subsets to detect the edge

domination, tiny edges and vertex domination and delete them from the instance. If the result of a branch is a "No" solution, the procedure *Branch* backtracks by deleting the element from the given instance without placing it in the hitting set. *adjust_couple* and *set_HS_o* procedures delete an element of status 'm" from all subsets of size three and of size two. It is worse noticing that when deleting an element from a subset of size three, the subset will be of size two and when deleting an element from a subset of size two, the subset will be of size one and the element that is in the subset of size one will be placed in the hitting set by applying the reduction rules of Fernau. After deleting the element whose status in array HS is equal to "m", the status of the element whose status is "m" is changed to "o". Then, the reduction rules described in chapter 4 are applied to the resulting subsets to detect the edge domination, tiny edges and vertex domination and delete them from the instance. And then the Branch procedure is reapplied till we reach a solution or No solution. In fact, if the instance has subsets of size two or subsets of size three and $k \leq 0$ then obviously the instance has No solution and if the instance has neither subsets of size three nor subsets of size two and $k \geq 0$ the instance has a solution which the elements that have status "i" in the array HS. *Check_status* procedure checks if the instance has subsets of size three, subsets of size two, has no subsets or has subsets of size three and of size two.

check_status(degree)

Input : degree and n

Output : status

The procedure checks whether the input instance has subsets of size three, subsets of size two, has no subsets or has subsets of size three and of size two. This is done by reading the array degree and checking the first column and the second column of array degree that represents the occurrence of each element in subsets of size three and subsets of size two respectively. Hence, *status* may have the following value:

0 : The instance has both subsets of size two and subsets of size three

1 : The instance has neither subsets of size three nor subsets of size two

2: The instance has only subsets of size two

3: The instance has only subsets of size three

4: The instance has only one subset of size three

5: The instance has only one subset of size two

get_instance(degree,pair,n_pairs,HS)

Input : degree, pair, n_pairs, HS and n

Output : x

The procedure mainly selects an element x on which we have to branch. According to Fernau, we have to check first subsets of size two and for each element in the subsets of size two, get the element x that has the maximum value for the following equation: degree of x in subsets of size three – degree of x in subsets of size two. If there are no subsets of size two, we take the element x that has the maximum degree. This procedure calls the *check_status* procedure to check if the instance has subsets of size two, subsets of size three, both subsets of size two and subsets of size three or no subsets.

set_HS_i(HS,degree,pair,n_pairs)

Input : element, degree, pair, n_pairs, HS and n

Output : element, degree, pair and n_pairs

The procedure deletes an element x whose status in the array HS is equal to "n" and set its degree to 0 in both columns of the array degree. In fact, the array element is checked for the occurrence of x in the following manner: if a couple in the array occur of the array element has one of its elements x, it is swapped by the couple that is at the end of the array occur and DEG is decremented by one. In this manner, the array element remains unchanged throughout the program and restoring it needs a restore of the array degree and HS only. For this reason, the array element is declared as global parameter and is not passed as parameter between function during the *branch* procedure. On the other hand, also this procedure checks the array pair of size n_pairs and delete all the occurrences of x from it. This results in an array new_ pair with smaller size. The array pair is then set equal to new_pair array.

simple_cases(HS,n_pairs,pair,k,degree)

Input : element, degree, pair, n_pairs, HS, k and n

Output : element, degree, HS, k, pair and n_pairs

This procedure checks the instance for the reduction rules of Fernau. It checks for edge domination, tiny edges and vertex domination. If vertex domination is detected in a subset of size three, the resulting subset of size two resulting from deleting the element without placing it in the hitting set will be added to the array pair. If the vertex domination is detected in a subset of size two, the resulting element of the subset of size one is placed in the hitting set and deleted from the array element and pair. It is worse noticing that when applying the vertex domination first, the edge domination will be automatically executed.

adjust_couple_branch(x,degree,HS)

Input : couple, no, HS, degree, element, n, pair and n_pairs

Output : couple

This procedure modifies the fourth column of the array couple. In fact, all subsets of size three where an element x occurs and whose status in HS is "m" will be subsets of size two after deleting x from them. The fourth column of the array couple is set to 2 for each couple that is equal to the resulting subset of size two.

set_HS_o_branch(HS,degree,n_pairs,pair,k,x,nocouple2)

Input : element, degree, pair, n_pairs, HS and n

Output : element, degree, pair and n_pairs

This procedure deletes an element x whose status is "m" from the array element and array pair without placing it in the hitting set. In fact, the degree of x is set to 0 in both columns of the array degree. In order to delete the occurrence of x from the array element, the array occur of the array element is checked for x, then the couple where x occurs is swapped with the couple that resides at the end. Concerning the array pair, it is checked for the occurrence of x and this results in a subset of size one whose element should be put in the hitting set.

```
branch_flag = 0;
reduction_flag = 0;
status = check_status(degree);
if ((status == 3) || (status == 5)){
        maxco = get_maxco();
        if (old_maxco == 0)
                old_maxco = maxco;
        if (old_maxco == maxco){
                ref     = check_maxco(maxco,degree);
                PK      = init_PK(maxco);
                fill_PK(PK,maxco,ref,degree);
        }
        else{
                reduction_flag =
                check_pseudo_reduction(HS,k,degree,old_maxco) ;
                if (reduction_flag == 1){
                        new_pair = pseudo_reduction(HS,k,degree,pair,n_pairs,old_maxco);
                        pair = new_pair;
                        k1 = *k;
                        status = check_status(degree);
                        if ((status == 3) || (status == 5)){
                                ref     = check_maxco(maxco,degree);
                                PK      = init_PK(maxco);
                                fill_PK(PK,maxco,ref,degree);
                        }
                }/* if (reduction_flag == 1)*/
                else {
                        edge_flag = check_edges(degree,old_maxco,k,HS) ;
                        if (edge_flag == 0){
                                ref     = check_maxco(maxco,degree);
                                PK      = init_PK(maxco);
                                fill_PK(PK,maxco,ref,degree);
                }
```

Figure 4 - *Branch_PK* Procedure (cont)

```
if (((status = 1) and (k1 ≥ 0)) {
        Print ("Congratulations you have done\n");
        Print the results
        branch_flag = 0 ;
}
else
if (((status = 0) or (status = 2) or (status = 3) or (status = 4) or (status = 5)) and
   (k1 = 0)) or
   (((status = 1) or (status = 2) or (status = 3) or (status = 4) or (status = 0) or
   (status = 5)) and (k1 < 0)))
                branch_flag = 1;
else{
We take a backup of the following parameters: k, pair, n_pairs, HS, PK and
 degree
if ((status = 3) || (status = 5))
        x       = PK[0];
        HS[x]   = 'n';
        new_pair = set_HS_i(HS,degree,pair,n_pairs);
        pair    = new_pair;
        HS[x]   = 'i';
        k = k -1
        new_pair = simple_cases(HS,n_pairs,pair,k,degree);
         pair = new_pair;
}
else{
        x = get_instance(degree,pair,n_pairs,HS);
        HS[x] = 'n';
        new_pair = set_HS_i(HS,degree,pair,n_pairs);
        pair = new_pair;
        HS[x] = 'i';
        k = k -1
        new_pair = simple_cases(HS,n_pairs,pair,k,degree);
        pair = new_pair;
}
```

Figure 4 - *Branch_PK* Procedure (cont)

```
if (branch(degree,pair,n_pairs,HS,k,0,maxco,reduction_flag) = 1){
        branch_flag = 0;
        We restore the following parameters: k, pair, n_pairs, HS, PK and degree
        if ((status = 0) or (status= 2) or (status = 4)){
                k1 = *k;
                HS[x] = 'm';
                get_no(HS,degree);
                couple = init_couple();
                fill_couple(pair,n_pairs,degree,HS);
                nocouple2 = adjust_couple_branch(x,degree);
                new_pair = set_HS_o_branch(HS,degree,n_pairs,pair,k,x,nocouple2);
                pair = new_pair;
                new_pair = simple_cases(HS,n_pairs,pair,k,degree);
                pair = new_pair;
                branch_flag = branch(degree,pair,n_pairs,HS,k,0,maxco,reduction_flag);
        }/*if */
        else {
                k1 = *k;
                HS[x] = 'm';
                get_no(HS,degree);
                couple = init_couple();
                fill_couple(pair,n_pairs,degree,HS);
                nocouple2 = adjust_couple_branch(x,degree);
                new_pair = set_HS_o_branch(HS,degree,n_pairs,pair,k,x,nocouple2);
                pair = new_pair;
                new_pair = simple_cases(HS,n_pairs,pair,k,degree);
                pair = new_pair;
                k1 = *k;
                y       = PK[1];
                HS[y]   = 'n';
                new_pair = set_HS_i(HS,degree,pair,n_pairs);
                pair    = new_pair;
                HS[y]   = 'i';
                k1      = *k;
                k1--;
                *k = k1;
                new_pair = simple_cases(HS,n_pairs,pair,k,degree);
                pair = new_pair;
```

Figure 4 - ***Branch_PK*** **Procedure (cont)**

```
if (branch(degree,pair,n_pairs,HS,k,0,maxco,reduction_flag) == 1){
        branch_flag = 0;
        We restore the following parameters: k, pair, n_pairs, HS, PK and degree

        HS[x] = 'm';
        get_no(HS,degree);
        couple = init_couple();
        fill_couple(pair,n_pairs,degree,HS);
        nocouple2 = adjust_couple_branch(x,degree);
        new_pair = set_HS_o_branch(HS,degree,n_pairs,pair,k,x,nocouple2);
        pair = new_pair;
        new_pair = simple_cases(HS,n_pairs,pair,k,degree);
        pair = new_pair;

        HS[y] = 'm';
        get_no(HS,degree);
        couple = init_couple();
        fill_couple(pair,n_pairs,degree,HS);
        nocouple2 = adjust_couple_branch(y,degree);
        new_pair = set_HS_o_branch(HS,degree,n_pairs,pair,k,y,nocouple2);
        pair = new_pair;
        new_pair = simple_cases(HS,n_pairs,pair,k,degree);
        pair = new_pair;

        if ( maxco <= k1){
                max_flag = 0;
                for(i=2;i<(maxco+2);i++){
                        z       = PK[i];
                        HS[z]   = 'n';
                        new_pair = set_HS_i(HS,degree,pair,n_pairs);
                        pair    = new_pair;
                        HS[z]   = 'i';
                        k--;
                        new_pair = simple_cases(HS,n_pairs,pair,k,degree);
                        pair = new_pair;
                }/* for(i=2;i < (maxco+2);i++)*/
        }/* if ( maxco <= k1)*/
else
        max_flag = 1;
 branch_flag =  branch(degree,pair,n_pairs,HS,k,max_flag,maxco,reduction_flag);
```

Figure 4 - ***Branch_PK*** **Procedure.**

Branch_PK(degree, pair,n_pairs, HS, k, max_flag,maxco,reduction_flag)

Input : m, n, element, no, couple, degree, pair, maxco, n_pairs, max_flag, reduction_flag and HS

Output : No if there is no solution of size at most k
Yes if there is a solution of at most k and print the solution

The *Branch_PK* procedure is the main procedure in the implementation of the hybrid algorithm. This procedure uses all procedures described in the previous section. In addition, it uses six new procedures to apply the branching, the reduction rule and checking of edges described in chapter 4. These new procedures are: *get_maxco, check_maxco, init_PK, fill_PK, check_pseudo_reduction* and *check_edges*. This procedure starts by calling the procedure *get_maxco* and getting the highest co-occurrence of two elements x and y in the instance. The parameter maxco contains the value of this highest co-occurrence. Then, it fills the array PK with x, y, $z_1, z_2, \ldots z_{maxco}$ by calling the procedure *fill_PK*. $z_1, z_2, \ldots, z_{maxco}$ are the neighbors of x and y in subsets of size three. The branching of the hybrid algorithm is as follows: x, y or $z_1, z_2, \ldots, z_{maxco}$. At each branch we check for the reduction rule by calling the procedure *check_pseudo_reduction*. If the condition is satisfied, the resulting elements are placed in the hitting set. If the condition is not satisfied, we check the number of edges by calling the procedure *check_edges*. If the number of edges are more than $(comax - 1) * k^2$ then the program returns directly a No solution otherwise, it continue the branching.

get_maxco(maxco)

Input : no and couple

Output : maxco

This procedure gets the highest co-occurrence of the instance by going over the column three of the array couple and checks for the maximum value.

check_maxco(maxco)

Input : maxco, couple and no

Output : reference

This procedure checks whether there is other couple that co-occur in maxco subsets of size three. It returns the index of the couple in the array couple.

init_PK(maxco)

Input : maxco

Output : PK

This procedure initiates the array PK knowing that its size is (maxco +2).

fill_PK(PK,maxco,ref,degree)

Input : PK, maxco, reference, n and degree

Output : PK

This procedure fills PK array by the couple that have maxco co-occurrence say x and y. In addition, by going over the array element, we fill the neighbors of x and y which are $z_1, z_2, \ldots, z_{maxco}$ also in PK array to get finally an array PK of size maxco + 2 .

check_pseudo_reduction(HS,k,degree,maxco)

Input : HS, K, degree, n, and maxco

Output : element, degree, pair and n_pairs

This procedure checks if the pseudo_reduction rule is satisfied. If it is satisfied, for each element, we call the procedure *set_HS_i* described above to place the element in the hitting set.

check_edges(degree,maxco,k)

Input : degree, n, maxco and k

Output : flag

This procedure checks if the number of edges in the instance where there is no element that occurs in more than (maxco -1) * k and that has only subsets of size three is greater than (maxco -1) * k^2. If it is greater than (maxco -1) * k^2, the value of flag is set to 1 and 0 otherwise.