UNIVERSITÄT HAMBURG

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

UNIVERSITAT AUTÒNOMA DE BARCELONA

Large Erasmus Mundus Consortium "MathMods"

Joint Degree of Master of Science in
Mathematical Modelling in Engineering: Theory, Numerics, Applications

In the framework of the
Consortium Agreement and Award of a Joint/Multiple Degree 2013-2019

Master's Thesis

Advanced Branching Rules for Maximum Stable Set Integer Programs

Supervisor:
Prof. Stefano SMRIGLIO
Co-Supervisor:
Prof. Fabrizio ROSSI

Candidate:
Fonyuy Theophile, FONDZEFE
Matricola: 238914

2015/2016

Laurea Magistrale in Ingegneria Matematica
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica
Università degli Studi dell'Aquila

Declaration of Authorship

I, Fonyuy Theophile, FONDZEFE, declare that this thesis titled, "Advanced Branching Rules for Maximum Stable Set Integer Programs" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"The value of an education in a liberal arts college is not the learning of many facts, but the training of the mind to think something that cannot be learned from textbooks."

Albert Einstein

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Abstract

Università degli Studi dell'Aquila
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Masters of Science

Advanced Branching Rules for Maximum Stable Set Integer Programs

by Fonyuy Theophile, FONDZEFE

The Maximum Independent Set (MIS) problem is one of the widely known NP-hard optimization problems. Unlike other NP-hard problems, there are no $\frac{1}{n^{1-\delta}}$-approximation algorithms for finding the maximum independent set of a graph. For this reason, the Branch-and-Cut algorithm is very useful for solving the mixed integer formulation of the MIS problem. The algorithm repeatedly creates and solve subproblems (same problem on subgraphs) that are relatively smaller in size than the original(parent) problem. As a recursive and resource intensive algorithm, it involves several major steps that are repeated for each subproblem until a certified solution is found. At the so called branching steps, one of many fractional variables is chosen and the subproblems will be generated by assigning to it an integer value. It is of utmost importance to choose this fractional variable carefully, because this in turn affects the number of subproblems that are created and solved before an optimal solution is found. Also, the choice of the variable determines the difficulty of the subsequent problems generated. We present a branching strategy that enhances the performance of the branch-and-cut algorithm by branching on violated odd cycles. Holes are a subset of odd cycles. Graphs whose subgraphs and their complements do not induce holes or antiholes are called perfect graphs. The maximum independent set is solvable in polynomial time for perfect graphs. The validity of the approach is empirically verified with the aid of DIMACS benchmark problems.

Acknowledgements

I like to acknowledge the guidance and support I have received from my supervisor Prof. Stefano Smriglio, and co-supervisor Prof. Fabrizio Rossi and other lecturers and administrative staffs at the University of L'Aquila (Italy) and the University of Hamburg (Germany) during these 2 years. This work would not have been possible without any of you.

In a similar light, thanks goes to my family and extended families for their moral and financial support during this period. There are yet not enough words to express my profound gratitude.

I am equally grateful to all friends, old and new, who have been there during these tough times. I will remember both the ones that have been kind and those that were difficult, you all make me a better person.

And above all, I am grateful to God Almighty for His unfailing love, and for putting all these people together in strategic places to complete this work.

Contents

List of Figures

List of Tables

List of Abbreviations

BnC	**B**ranch-(and)-**C**ut
DIMACS	Center for **Di** **M**athematics and Theoretical **C**omputer **S**cience
ILP	**I**teger **L**inear **P**rogramming
LP	**L**inear **P**rogramming
MILP	**M**ixed **I**teger **L**inear **P**rogramming
MIS	**M**aximum **I**ndependent **S**et
NP	**N**on-deterministic **P**olynomial

List of Symbols

$\alpha(G)$ cardinality of Maximum Independent Set of a graph G

Dedicated to the likes of Bernard Fonlon, and those who would not be deceived by pseudohistory.

Chapter 1

Background

1.1 Introduction

As sciences advances, the complexity of the problems faced by both engineers and scientists increases more or less proportionately. Today, many researchers and industries are gradually shifting from humans (man power) to machines for both labour intensive and computationally sophisticated tasks. These machines themselves rely on algorithms written by other humans or generated by other machines. As the complexity of the problems increases, the resources required to solve even the small instances becomes less affordable. Moreover, many everyday real life instances of these problems are not easy. Some are the most difficult case scenarios of the problem type. More efficient and simple algorithms are thus highly needed in almost every scientific discipline. Over the last few decades, computational problems and their corresponding algorithms - where available - have gained a lot of popularity. For some of these problems, even efficient(or polynomial time) algorithms have been developed. But for many other problems, no polynomial time algorithm is known. Many are known to belong to the well known *Non-deterministic Polynomial* (*NP*) class of problems. One can be formulate these problems as discrete optimization problems on graphs. We begin by presenting a brief overview of the terminology in graph theory that will be used in the subsequent chapters.

1.2 Simple Finite Undirected Graphs

Graphs are abstract structures which are often used in discrete mathematics to model pairwise relationships between objects. They are commonly described as ordered pairs of the form **G** = (**V**,**E**), where **V** is a set of vertices, nodes or points and **E** is a set of arcs, edges or lines connecting the vertices. If the elements of **E** have no specified direction, then the graph is said to be *undirected*. And if the number of elements in **V** are finite in number, then the graph is *finite*. It is also possible to have a graph with arcs (called loops) that connect a point to it self, as well as one with multiple edges between two points. A *simple* graph is one that does not contain any multiple edges and loops. Throughout remaining part of this work, we will use the word graph to mean a simple finite undirected graph. Also, the letter **V** will be used to denote set **V** = $\{v_0, v_1, v_2, ...v_{n-1}\}$, where n is cardinality of the set. Where an arc or line connects two distinct elements of **V**, say v_i and v_j, 1$\leq$ i, j $\leq$ n, then the tuple (v_i, v_j) represents an element of **E** and the two points are said to be adjacent. A graph is said to be *empty* if it has no edges. It is said to *complete* if every point is connected to every other point. A subgraph

that is complete is also called a *clique*, and it is said to be *maximal clique* if no other node from the graph can be added to the clique to form a bigger clique. Also, the *degree* of vertex, v, denoted *deg(v)*, is the number of nodes adjacent to that vertex in a graph. Thus in an *empty* graph, the degree of every vertex is zero, and in a complete graph on n nodes, the degree of every vertex is *n-1*.

1.3 Maximum Stable Set Problem

A *stable set* is a subset of the set of points such that every induced subgraph graph is empty, that is no two vertices are adjacent. It is also known as an *independednt set*. It will be evident that each arc or edge will have only one or none of its endpoints in an independent set, and a graph can contain many of such sets. Such a set is said to be *maximal* if it is not a subset of another independent set. A graph can has many maximal independent sets of varying cardinalities. The cardinality of largest independent set of a graph **G**, is known as the *independence number* of **G**, denoted α**(G)**.

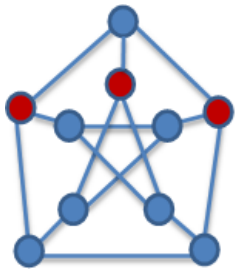

FIGURE 1.1: Red colored vertices form a maximal independent set for the Peterson graph.

Any independent set **S**, of cardinality α**(G)** is called a Maximum Independent Set (MIS) of **G**. he problem of finding such a set for any given graph, is referred to as the MIS problem. Like the Traveling Salesman Problem, it is one of the most important 'classical' combinatorial optimization problems. It is also known as the *vertex packing*, *coclique* or *stable set* problem. Today, no polynomial time algorithm is yet known that finds a maximum independent set on an arbitrary graph. A problem is said to be solvable in polynomial time if there exist an algorithm that solves any instance of the problem and the time taken by the algorithm in the worst case is a polynomial function of the size of the problem. Otherwise, such a problem is said to be *NP-hard*. Besides being an *NP-hard* optimization problem, the MIS problems is also hard to approximate (Rebennack et al., 2011; Arora and Safra, 1992). The MIS problem is not only theoretically valuable by has applications in several fields. For some applications of the maximum independent set problem in fields like information retrieval, signal transmission analysis, classification theory, economics, scheduling, and biomedical engineering see (Butenko, 2003). The MIS problem is not in general difficult for all types of graphs; it is solvable in polynomial time for a class of graphs *perfect graphs*.

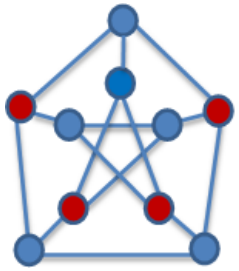

FIGURE 1.2: Red colored vertices form a maximum independent set for the Peterson graph.

1.4 Odd cycles, Holes, and Perfect Graphs

Consider a graph **G**, then a sequence of k distinct vertices of the graph $v_1, v_2, ..., v_k$ having no other edges between them but for the edges (v_k, v_1) and (v_i, v_{i+1}), i=1,2, ... k-1, is called a *chordless cycle*. A *hole* is a chordless cycle on five or more vertices - that is k$\geq$5. An *antihole* is the complement of a hole. (Nikolopoulos and Palios, 2004). An example of a hole and an antihole are shown below. Note that antihole with five nodes is isomorphic to an odd hole with five nodes.

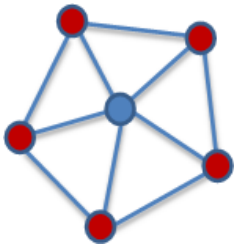

FIGURE 1.3: The 5 red vertices induce a hole..

FIGURE 1.4: An antihole of size 7.

Holes and antiholes are useful in the identification of a class of graphs called *perfect graphs*. By the *strong perfect graph theorem*, a graph **G** is perfect if and only if **G** does not contain any odd circuit (or hole) of length at least 5, or its complement, as an induced subgraph (Chudnovsky et al., 2006; Grotschel, Lovasz, and Schrijver, 1980). As shown in Figure 1.5, the Peterson graph is not perfect. It contains at least two holes induced by the green coloured vertices and the red coloured vertices. We claim that, though not every graph is perfect, every graph contains a perfect induced

subgraph. As one removes some of the vertices of a *non-perfect* graph, the induced subgraph can become perfect. If the vertices that are removed first, are those that break the odd circuits, then one attains a perfect induced subgraph with just a few points removed. It is our assumption that one has to remove only as many vertices as the number disjoint odd holes and/or antiholes. This property of perfect graphs makes it possible to solve some NP-hard combinatorial optimization problems in polynomial. It is worth considering the possibility of an algorithm that benefits from this property. Checking if a graph is perfect can be done in polynomial time, but the algorithms are quite sophisticated. Some of the best known algorithms have a running time of $O(|V|^6)$, (Rebennack, Reinelt, and Pardalos, 2012).

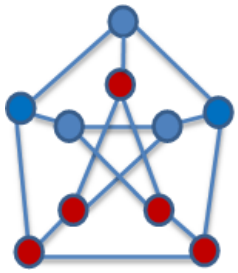

FIGURE 1.5: Similarly coloured vertices form a hole.

1.5 Integer Programs

Integer Programming (*IP*) problems are mathematical optimization problems in which the variables are restricted to be integers. This term is often used interchangeably with *Integer Linear Programming*, in which the objective function and the constraints of the problem are linear. The generic form of an *IP* problem is as follows

$$\begin{aligned} &\textit{maximize} && c^T x \\ &\textit{subject to} && Ax \leq b, \ \ \forall\, (v_i, v_j) \in \mathbf{E} \\ & && x \in Z^n. \end{aligned}$$

with integral decision variables x. The objective here is to find a vector $x \in Z^n$ that maximize the function $c^T x$ while satisfying the linear constraints $Ax \leq b$. A special case of Integer Programs, is the *Zero-One* linear programming. Here the variables are restricted to be either 0 or 1. The integrality constraints of in IP makes the problems *NP hard*. When the integrality constraint of a ILP problem is replaced by a non-negativity constraint, we obtain a *Linear Programming(**LP**)* problem. The new problem is called a *LP relaxation* of the former. LP problems can be solved in polynomial time (Rebennack, Reinelt, and Pardalos, 2012). There exist some algorithms that find exact solutions to integer programs. In particular we have the so called *Branch-and-Cut (BnC)* algorithms. This is a combination of the so called Branch-and-Bound algorithm and cutting plane like those of Rossi and Smriglio (2001).

Also important for the understanding of this work are a few polyhedral terminologies. We define a polyhedron as the solution set of a system

of linear inequalities. Whenever such a solution set is bounded, then it is referred to as a *polytope*. A linear inequality $\alpha^T x \leq b_0$ is thus valid with respect to a polyhedron P, if P is a subset of {x | $\alpha^T x \leq b_0$}. Let F $\subset$ of P such that there is a valid inequality $\alpha^T x \leq b_0$ for P with F = $x \in P | \alpha^T x \leq b_0$ and the inequality is not dominated by any other valid inequality. Then F is called a *facet* of P and the inequality is called a *facet-defining* inequality for P. Also, the convex hull of points $y_1, y_2, ..., y_n$ is $\in R^d$ is the set of points x satisfying x = $\sum_{i=1}^{n} \lambda_i y_i$ with $\sum_{i=1}^{n} \lambda_i = 1$ and $\lambda_i \geq 0 \forall i$. We denote this as *conv{*$y_1, y_2, ..., yn$*}* (Rebennack, 2008). As we shall see later, the convex hull of vectors associated to the stable set polytope concides with the polytope of the inequalities of the so called clique formulation of the stable set of a graph.

1.6 Branch-and-cut Algorithms

Many attempts have been made at developing algorithms that solve the ILP problems for arbitrary graphs. Unfortunately, the worst-case complexity of many classical algorithms is exponential function of the problem size. Robson algorithm developed in 1985 for the finding the MIS in an arbitrary graph has complexity $O(2^{2.786n})$ (Robson, 1986). Combinatorial algorithms like the branch-and-cut are now the most used and successful tools for solving ILPs. The algorithm iteratively solves a set of LP relaxations of the MIP problem. What follows are the main subroutines of the branch-and-cut optimization algorithm.

- A preprocessing routine that tightens the user-supplied formulation
- A heuristic that yields *good* integer feasible solutions quickly
- A cut generation procedure that tightens the linear programming relaxation.
- A branching strategy that selects the *next* branching variable and determines the search tree.

The branch-and-cut algorithm begins by solving a linear relaxation of the given problem using the *simplex* algorithm. If the optimal solution obtained contains fractional values for variables that were supposed to be integers, then a cutting plane algorithm may be used to tighten the relaxation of the problem to a better approximation of the MIP problem. Then a branch-and-bound algorithm is use to solve the problem via a divide and conquer approach. That is, it divides the problem into two subproblems, for which the LP relaxations are solve again and the optimal solutions are checked for fractional variables and the process repeats. Evidently, there may be many possibilities of how to split into the new subproblems. So it becomes important to understand how splitting affects the overall performance of the algorithm. During the branching process, non-integral solutions to LP relaxations serve as upper bounds and integral solutions serve as lower bounds. Another important property of the BnC algorithms is that, they do not only return a final solution to a problem, but also provide a measure of the quality of the solution. Such certificates of optimality are the main basis of confidence in the solution return. A flowchart of a branch

and cut algorithm for a maximization problem is shown in Appendix A. For a more elaborate discussion of branch and cut algorithm for the maximum stable set problem, see Rebennack et Al (Rebennack, Reinelt, and Pardalos, 2012). Our main interest in this project goes to the last main routine above; the branching strategy. We develop a new branching rule and evaluate it performance by testing it on some DIMACS benchmark problems.

1.7 Branching Strategies

As a very important part of the Branch-and-Cut algorithm, a lot of research on this algorithm goes towards improving the branching rule. Usually, the quality of any rule is measured by the change in the objective value of the LP relaxation of the two subproblems compared to that of the parent problem (Achterberg, Koch, and Martin, 2005).There are a wide variety of branching heuristics. In particular, we focus on those that practically involves choosing a variable x_i with a fractional value in the optimal solution of the current LP relaxation. And we rather evaluate our branching rule the number of nodes explored during the optimization process. We discuss some of already well known rules briefly below.

1.7.1 Most Infeasible Branching

This is a very popular heuristic that chooses a variable with the fractional part closest to 0.5. The intuition here is to select the variable where the least tendency can be recognized to which direction (up or down) the variable should be rounded. *Achterberg et al* have shown that this is no better than selecting the variables randomly (Achterberg, Koch, and Martin, 2005).

1.7.2 Pseudo Cost Branching

This strategy works by keeping a history for each variable x_i, of the change to the objective function when it was previously chosen as a branching variable. The candidate variable that have caused the most change on the objective function is then chosen. There are many variations of this rule have been develop. Unfortunately, the rule is uninformative at the beginning of the branch-and-bound algorithm since most variables have not been used for branching.

1.7.3 Strong Branching

The idea here is to choose the a few candidate variables that gives the most improvement on the objective function. This is done by temporarily introducing lower bounds and/or upper bounds to each of the chosen variables and solving the LP relaxations. The number of candidate variables tested is decided by some paramter.

1.7.4 Full Strong Branching

As a variation of the strong branching, this heuristic tests all candidate variables. It becomes computationally expensive as the number of candidates increases. Attempts to reduce the computational cost include methods like

not completely solving the LP relaxations for a subset of the candidate variables.

1.7.5 Reliability Branching

This is the rule proposed by *Achterberg et al*. The strategy combines the idea of pseudocost branching and strong branching. It uses strong branching both on variables with uninitialized pseudocost values and those with unreliable pseudocost values. A reliability parameter is used to determine if pseudo-cost values are reliable (Achterberg, Koch, and Martin, 2005).

The above rules are general branching rules implemented in most optimization solvers. However, for solving the maximum independent set, it has been empirically shown by Carraghan and Pardalos (1990) that by branching on nodes with a high degree, the size of the tree can be substantially reduced (Rebennack, Reinelt, and Pardalos, 2012).

1.8 Problem Statement

The goal of this work is to present a branching rule that enhances the performance of the branch-and-cut algorithm. With the aid of DIMACS benchmark problems, we empirically evaluate the approach using the IBM ILOG CPLEX optimizer. In effect, we present results showing that with the proposed branching strategy, the number of nodes evaluated by the Branch-and-Bound algorithm is significantly reduced.

Our claim therefore is that, by branching on fractional variables that belong to violated odd cycles, we are able to arrive subgraphs that are perfect in a shorter time, and thus the LP relaxations will provide better solutions (smaller upper bounds).

"Any physical theory is always provisional, in the sense that it is only a hypothesis: you can never prove it. No matter how many times the results of experiments agree with some theory, you can never be sure that the next time the result will contradict the theory... Each time new experiments are observed to agree with the predictions, the theory survives and our confidence in it is increased; but if ever a new observation is found to disagree, we have to abandon or modify the theory", (Hawking, 2011).

Chapter 2

Integer Program Formulation and The Branching Rule

2.1 Integer Linear Programming (ILP) formulation

As with many graphs problems, the MIS problem can be formulated as Integer Linear Programming (ILP) problem. Formulating the MIS problem as a MIP problem is important because it can then be solved by the combinatorial algorithms such as branch-and-cut method, which is an exact algorithm consisting of cutting plane method and a branch-and-bound algorithm (E., 2002). By letting the variables $x_1, x_2...x_n$ represent the vertices of a graph *G* = (*V*,*E*). Such that each variable x_i define

x_i = 1 if the vertex v_i is part of the independent and x_i = 0 otherwise. Then we must find the vector $x = (x_1, x_2...x_n)$ that

$$\begin{aligned} \textit{maximize} \quad & \sum_{i=1}^{n} x_i \\ \textit{subject to} \quad & x_i + x_j <= 1, \ \ \forall\, (v_i, v_j) \in \mathbf{E} \\ & x_i \in \{0, 1\}. \end{aligned}$$

We observe that this formulation has only |E| constraints and |V| variables and restricts the variables x_i to be binary. So the MIS problem is indeed a 0-1 Integer Programming problem.

The stable set polytope of a graph **G**, is defined as the convex hull of all stable set vectors in G. It is denoted by,

$$P_{STAB}(G) := conv\{X^S | S \subset V \ stableset\},$$

where X^S is the incidence vector of set *S*. From the IP formulation above, we see that $P_{STAB}(G)$ is a polyhedron and since its bounded by the $|V|$-dimensional cube, it is also a polytope. The definition of a stable set implies that the unit vectors are always stable sets. Trivially, the zero vector is a stable set, the empty set, therefore, the stable set polytope is full-dimensional. Unfortunately, the binary constraints on x make it hard to solve this linear program. The LP relaxation of the MIS problem above can be solved efficiently (or in polynomial time), and has the unusual property that any variable that is interger in its optimal solution has the same integer value in some optimal solution to the 0-1 ILP problem above. Unfortunately, it is generally the case that an optimal solution to the LP relaxation of the edge formulation above has variable equal one-half. (Nemhauser and Sigismondi, 1992).

The LP relaxation of this problem is obtained by replacing the constraints

$x_i \in \{0, 1\}$ with the constraints $x_i \geq 0 \ \ \forall i = 1, 2, ..., n$

From this relaxation, we have the so called stable set polytope relaxation, denoted

$$P_{RSTAB}(G) := \{x \in R^n | x_i + x_j \leq 1, 0 \leq x \leq 1, \forall (v_i, v_j) \in E\}$$

It is evident from this that $P_{STAB}(G) \subset P_{RSTAB}(G)$. As an example, consider a complete graph with $n \geq 3$ vertices. The vector x with $\frac{1}{2}$ in each component belongs to $P_{RSTAB}(G)$ but not to $P_{STAB}(G)$, since the MIS for a complete graph has cardinality at most one. This example shows that the relaxation above is very weak and hence cannot be a good choice for a Branch-and-Cut algorithm for a general graphs (Grotschel, Lovasz, and Schrijver, 1980; Rebennack, 2008). There may be more than one ILP or Mixed Integer Programming (MIP) formulations to a mathematical optimization problem. The LP relaxations of these formulations may result in different solutions. Thus one formulation can be *better* or *stronger* than another if the LP relaxation always have a smaller upper bound. Another well-known better formulation for the MIS problem is based on the fact that at most one vertex in any maximal clique, which we denote as Q, of a graph can be part of the independent set.

$$\begin{aligned} \textit{maximize} \quad & \sum_{i=1}^{n} x_i \\ \textit{subject to} \quad & \sum_{x_i \in Q} x_i <= 1, \ \ \forall \, Q \in \mathbf{G} \\ & x_i \in \{0, 1\}. \end{aligned}$$

This formulation is known as the *clique formulation*. The constraint are called *clique constraints* and they ensure that any solution satisfying the integrality clause also satisfies the *edge constraints* used in the previous formulation. Thus, the a LP relaxation to the clique formulation in general will provide a much tighter bound than a LP relaxation to the edge formulation. Unfortunately, the LP relaxation of the clique formulation is NP-hard for general graphs (Nemhauser and Sigismondi, 1992). We define the *clique-constraint* stable set polytope as,

$$P_{QSTAB}(G) := conv\{x \in R^{|V|} | x_i + x_j \leq 1, 0 \leq x \leq 1, \sum_{x_i \in Q} x_i <= 1, \forall Q \in \mathbf{G}\}$$

We note that for a perfect graph, $P_{QSTAB}(G) = P_{STAB}(G)$, which means that the inequalities of P_{QSTAB} are sufficient to describe the stable set polytope. Another constraint not necessarily implied or dominated by the clique constraints above, is the so called *odd cycle constraint*. Infact, if $\mathbf{G}$ is an odd cycle, the vector $x = (\frac{1}{2}, \frac{1}{2}, ..., \frac{1}{2}) \in R^V$ also belongs to $P_{RSTAB}(G)$ but does not belong to $P_{STAB}(G)$. The odd cycle constraint requires that the sum of the values of the vertices in an odd cycle be at most the integer part of half the cardinality of the odd cycle. That is,

$$\sum_{x_i \in C} x_i <= \lfloor \frac{|C| - 1}{2} \rfloor \tag{2.1}$$

where C represents the set of points in an odd cycle of graph G. The solution of a linear relaxation contains a *violated odd cycle (VOC)* if this inequality is not satisfied for some odd cycle in the graph G. Since the odd cycle inequalities cannot be obtained from linear combinations of the clique inequalities, they can be used to tighten the LP relaxation of the clique formulation (Nemhauser and Sigismondi, 1992). In the subsequent paragraphs, we will show how this new inequality could be used to decide or choose the next branching variable of a branch-and-cut algorithm.

2.2 Branching Rule

Since our goal is to develop a branching strategy that leads us towards perfect subgraphs early on in the branch-and-cut tree, we have to choose our branching variable in such away that at least one odd hole or antihole is destroyed, or equivalently, set the value one or zero for at least one violated odd cycle inequality if any exsits. To be able to do so, one has to be able to identify odd holes (or odd cyles respectivelly) in a subgraph induced by the remaining fractional variables. However, instead of searching for holes and antiholes, we focus on violated odd cycles of length greater than or equal 5. Odd holes are a subset of odd cycles. Moreover, no odd cycle containing a node whose corresponding variable has value of 0 or 1 can violate the odd-cycle inequality (Rebennack, Reinelt, and Pardalos, 2012). Thus, by branching such that violated odd cycles are destroyed, we will have the same intended results whenever our odd cycle is an odd hole. On the other hand, this ensures that the number of violated odd cycles inequalities are increasingly reduced as the branch and cut algorithm progresses, thus the stable set polytope is closely represented by the clique formulation for the subgraphs (or subproblems) generated. In the following section we discuss the algorithm used to detect violated odd cycles.

2.3 Violated Odd Cycle

A computationally effective but not exact algorithm for detecting violated odd cyles was proposed by Hoffman and Padberg, see (Hoffman and Padberg, 1993). We begin this section by discussing this violated odd cycle detection algorithm, and afterwards we show how we modified it to ease implementation and speed up the process of finding these odd cycles. We used the graph in Figure 2.1 to demonstrate various intermediate and final results of the algorithms.

The odd cycle algorithm begins by choosing a vertex, say $v_1 \in V$ which is called the root node. Our preferred choice for such a root node, is a fractional vertex with the least degree and feasible for branching. The next step is to build a layered graph **L** starting from the root node while adding edges and nodes from the original graph **G**. Each level of **L** is defined by the *edge distance* from the root node. That is, the path from each level k to the root node v_1, has exactly k edges. So we have all the vertices adjacent to v_1 on

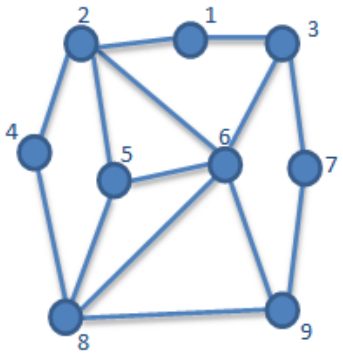

FIGURE 2.1: A sample graph

level one, and those adjacent to level one vertices except those already on same or lower level on level 2 and so forth. Below is the corresponding layered graph, with the vertex v_1 as the root node, for the graph in Figure 2.1.

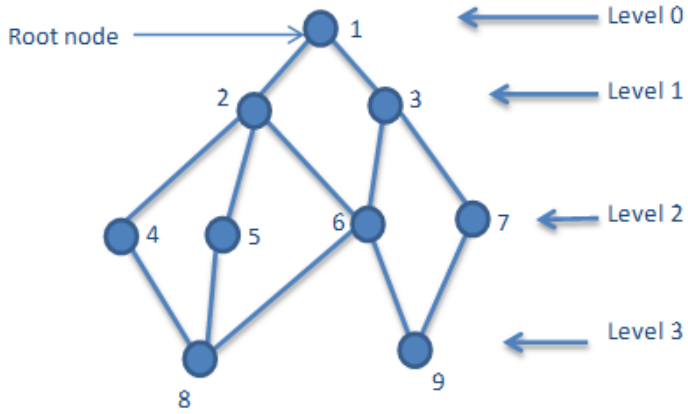

FIGURE 2.2: A layered graph for graph in Fig 2.1

By using the current LP relaxation solution as the values of the vertices, we are able to construct a layered graph with weighted edges. An edge, (v_i, v_j), in the layered graph has weight equivalent to $1 - x_i - x_j$. As an example we have provided below a sample of our graph in Figure 2.1 with weighted nodes satisfying the clique constraints presented in the beginning of this chapter. We have also shown the corresponding layered graph, together with the weights of the edges (v_1 is still the root node).

To find an odd cycle, one is required to choose two vertices, at the same level in the layered graph. The two vertices should be adjacent in the original graph. Also, since we are interested in odd cycles of length greater or equal to 5, the two vertices must be at a level greater than 1. Vertices at level 1 can only form odd cycles of length 3 according to the method we intend to follow. An odd cycle of length 3 is a *clique*. Such a cycle cannot violate the odd-cycle inequality since it is a subset of the clique constraints (or it is dominated by the maximal clique inequalities). The next step is

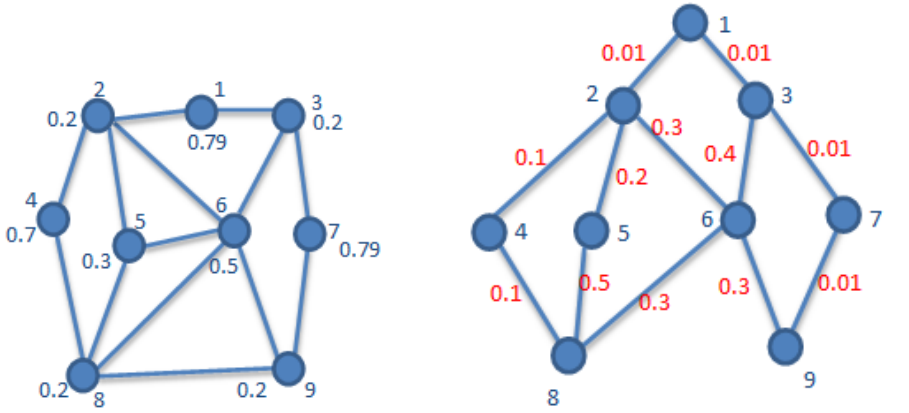

FIGURE 2.3: The weighted graph and the corresponding weighted layered graph

then find the shortest path from the root node to each of these vertices on the weighted layered graph. Since all weights of the layered graph are positive, *Dijstra's* algorithm is good enough for this purpose. The two paths found must not have any common edges and/or nodes besides the root node. To ensure that the two paths do not have any common edges, it is required that after finding the shortest path to one of the vertices, the edges in this path should be *'blocked'*. In other words, given *large* weights such that every other path from the root node to the second vertex not using the same edges is shorter. We use the following example to demonstrate the flow of the algorithm. Beginning at level 2, we observe that the only two vertices that are adjacent in **G** are the vertices v_5 and v_6. Thus no other pair of vertices in level can be used to find an odd cycle according to the description above. It is also important which vertex that we find the shortest path for first. Suppose we search first for the shortest path from v_5 to v_1 , we get the $[v_5, v_2, v_1]$, shown in green in Figure 2.4.

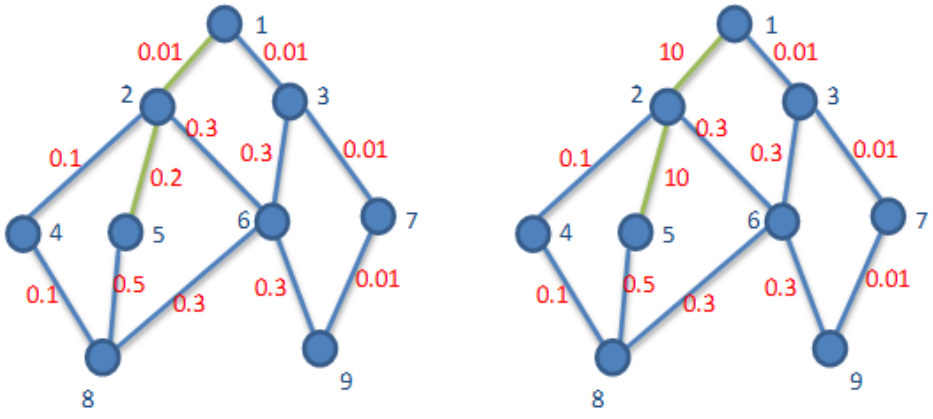

FIGURE 2.4: The shortest path to root node from v_5, and the corresponding *'blocked'* edges

Now, if the edges (v_1, v_2) and (v_2, v_5) of this path are *'blocked'* by increasing the corresponding weights of the edges to say 10, the shortest path from v_1 to v_6 becomes $[v_1, v_3, v_6]$ and not $[v_1, v_2, v_6]$.

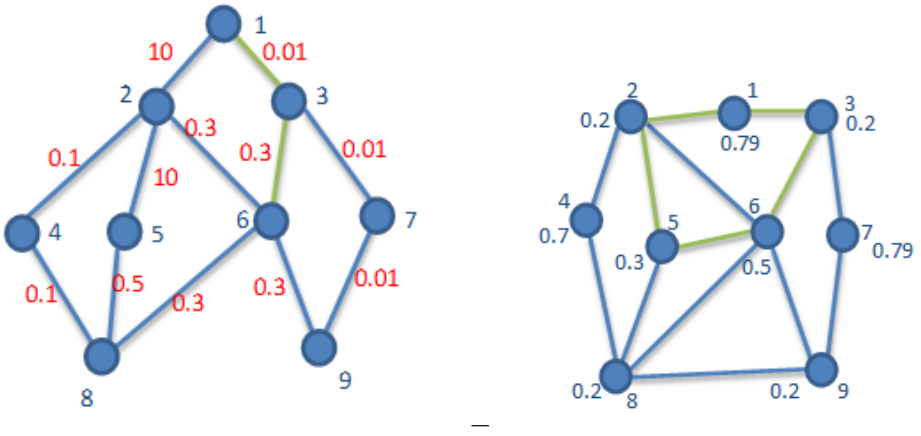

FIGURE 2.5: The shortest path to root node from v_6, corresponding odd cycle in sample graph

The path from the root node to both vertices does not contain any common vertices. Thus, satisfying the necessary conditions to form an odd cycle. The cycle is obtained by concatenating the two paths to form a new path in the graph **G** from v_5 through v_1 to v_6. The cycle is completed by adding the edge connecting the v_5 and v_6 to the new path. Thus $C = [v_5, v_2, v_1, v_3, v_6]$ induced an odd cycle in **G**. Since we have that

$$\sum_{x_i \in C} x_i = 0.69 + 0.3 + 0.5 + 0.3 + 0.3 = 2.09 > 2 = \lfloor \frac{5-1}{2} \rfloor = \lfloor \frac{|C|-1}{2} \rfloor \quad (2.2)$$

we found a violated odd cycle. Thus whenever we find an odd cycle, it is not necessary called a violated odd cycle unless the corresponding odd cycle inequality is violated by this odd cycle. It is important to note here that, if in the above process we started first with the vertex v_6, and the resulting path found was $[v_1, v_2, v_6]$, then there will be no disjoint path going from the root node to v_5. Also, the algorithm does not prevent the two paths from the root node to the chosen vertices from having common nodes besides the root node. For this reason, the resulting solution might indeed not be an odd cycle. For example, consider the two paths, $[v_1, v_2, v_6, v_8]$ and $[v_1, v_3, v_6, v_9]$ having no common edges but 2 common nodes, v_1 and v_6. They are not the shortest paths for the LP relaxation solution provided, but there is the possibility that an instance could occur that these are the shortest path. Concatenating the paths and connecting the vertices v_8 and v_9 as explained above, one does not get an odd cycle but a self-crossing path, see Figure 2.6. So it is necessary to *'block'* not only the edges belonging to the initial path from root node to the first vertex, but also the edges incident to any node along this path. Moreover, suppose that there were more than 1 distinct (not having any common vertices) pairs of adjacent vertices at the level two or higher. If the first pair of nodes in the layered graph does not contain a violated odd cycle, the original weights of the *'blocked'* edges

have to be recomputed before moving to the next pair of edges. This is repeated for each pair of adjacent vertices at any level until a violated odd cycle is found. As the size of the graph increases and the degrees of the nodes increases, the more the weight of the edges that have to be changed and recomputed. The algorithm is not exactly very scalable.

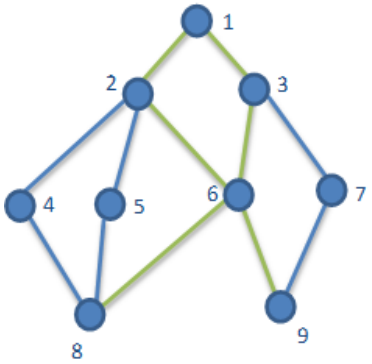

FIGURE 2.6: Two paths in layered graph having more than 1 common node

For these reasons, we modified the above odd cycle algorithm in the following way. We also begin by choosing a root node. However, instead of building a layered graph, we build a tree, in the sense that each vertex of graph **G** will have at most one parent. Again only nodes whose corresponding variables have value different from 0 and 1 are included in the tree graph. As we already mention, one single node with such a property implies that no odd cyle containing this node can violate the odd-cycle inequality.

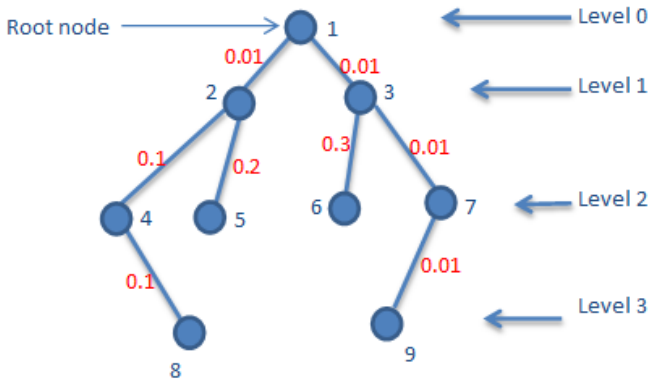

FIGURE 2.7: A tree graph for graph in Fig 2.1

Unlike in the layered graph where the vertices v_6 and v_7 had 2 and 3 parents respectively, the tree graph will associate to each node (except the root node), the parent with the largest corresponding variable value, x_i. This being the case, the tree generated from the sample graph in Figure 2.1, is shown in Figure 2.7. As this point, instead of looking for the shortest paths between nodes at a level 2 (or higher) and the root node, we look for the shortest path between two nodes at the same level in the tree. It is

still required that the two nodes be connected in the original graph **G**, and in addition, that the two nodes do not share the same parent node. Also, we can use the Dijkstra's shortest algorithm to compute this path. Since, the complexity of this algorithm is $O(|E|log|V|)$, the tree greatly reduces the number of edges and the fact that only nodes whose corresponding variable has value different from 0 and 1 helps to reduce the size of $|V|$ in tree. Thus, we have a speed up in the violated odd cycle algorithm.

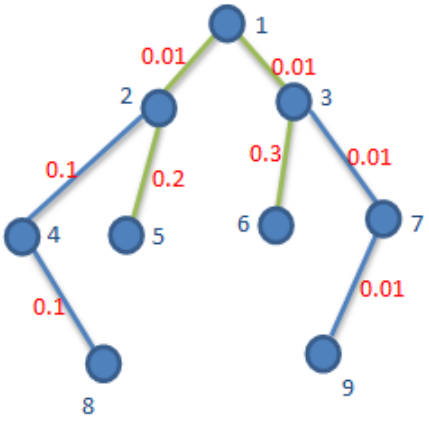

FIGURE 2.8: The shortest path from v_5 to v_6 in the tree

In both algorithms, if a violated odd cycle is not found by choosing a particular node as the root node, the process can always be repeated for a different candidate root node. As shown above, the resulting shortest path between the vertices v_5 and v_6 is the same that was found in the layered graph. But this time, without any intermediate changes to the weights of the edges. This modification solves the problem of common edges and nodes, by ensuring that a node can only be reach from the root node through one parent. It is also easy to implement.

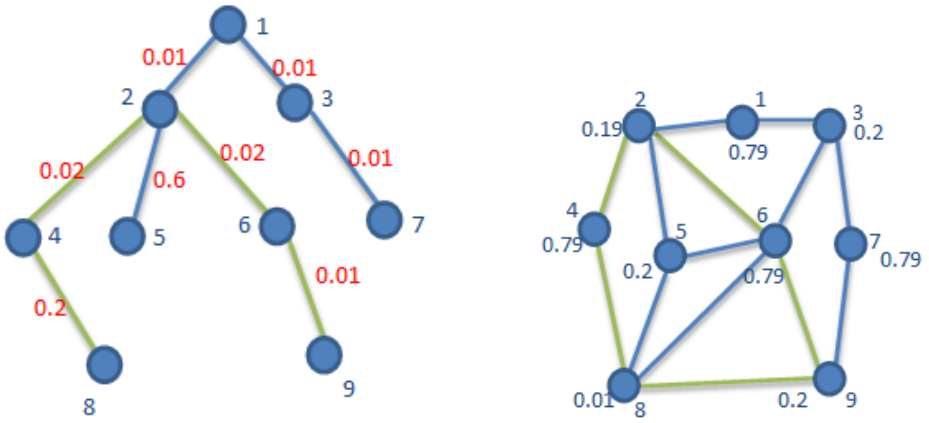

FIGURE 2.9: The shortest path from v_8 to v_9 in the tree does not include root node, and is odd cycle in **G**

Moreover, using the method of the layered graph presented before, one

is constraint us to finding paths that include the root node. Also, at a given level *k*, the only paths that can be found are those containing the root node and must be of length *2k+1*. Fortunately, this is not the case in the modified version. Consider the following possible tree. The shortest path between vertices v_8 and v_9 (shown in green) does not include the root node, and contains 5 nodes while the two vertices are at level 3. The advantage of having this, is that probability of finding a violated odd cycle is less dependent the root node chosen. Thus the tree based heuristic does not loose any chances of finding a violated odd cycle that could be done by the layered graph approach method but rather improves a lot its efficiency.

The above algorithm helps to find a violated odd cycle. To conclude that no violated odd cycle exist, one has to build the tree graph for every node as root node and find the shortest path between each pair of adjacent vertices at every level greater than or equal 2. This is time consuming, so besides considering only nodes whose corresponding variable has a fractional value, we propose to put a time limit to how long the algorithm uses to find a violated odd cycle. However, one could find minimum-weight odd cycle for the graph **G**, and if the solution to LP relaxation at the current node in the enumeration tree, satisfies the corresponding odd-cycle inequality, then all odd-cycle inequalities are satisfied.

2.4 Branching Variable

As mentioned earlier, this is not an exact algorithm for finding violated odd cycles in arbitrary weighted graphs but it is computationally effective. So at some points, we might not find violated cycles. Moreover, since it is time consuming to do this for all the nodes. It is there better to limit the algorithm to run only on a few nodes or for a limited time in order to make it time effective. If a violated odd cycle is found within this period, then it is returned. Otherwise, at the end of the time frame, the subroutine stops. Since we are using the IBM ILOG CPLEX Optimization Solver, we call the violated odd cycle algorithm from within the Branch Callback Class, and select our branching variable as follows. When a violated odd cycle is found, the variable in the odd cycle with the maximum degree in the graph **G** is chosen as the branching variable. Two branches are then created base on this variable, the *UP* and the *DOWN* branch. On the other hand, if a violated odd cycle is not found, a fractional variable that is feasible for branching with maximum degree is chosen. As already meantioned, by branching on with a higher degree vertices, the size of the tree is reduced.

At this point we discuss the basis and reason for how the first root node is chosen. First we point out that every odd cycle between vertices at level two, has the root node as one of it vertices. Since we choose our branching variable from amongst the variables in the odd cycle, we must ensure that the cycle has at least one variable that is fractional and feasible for branching. Also, it preferable to build a tree that grows deep rather than wide. Choosing a vertex with a large degree, will place more nodes on the first level of the tree and fewer nodes on the higher levels. We look for the odd cycles by considering vertices on the second and higher levels. Thus it is better to build a tree with the root node having fewer neighbors despite the independence of our algorithm from the choice of the root node.

Chapter 3

Empirical Validation

3.1 Introduction

In order to evaluate the performance our proposed branching rule, we implemented it in the Python. Since the time used We begin this chapter by presenting some of the software tools we use implement and evaluate the performance of our branching rule. We also discuss some parameters introduced to keep the algorithm time effective. Then we present a table of the results obtained from testing the branching rule on a computer with an AMD V120 Processor 2.2GHz, 3GB RAM x64-based processor.

3.2 Software tools

3.2.1 Optimization solvers

The development of Branch-and-Cut solvers was driven by Mannino and Sassano through their introduction of the edge projection. Rossi and Smriglio developed a *good* Branch-and-Cut algorithm with the aid of separation routines. Their algorithm still provides one of the best dual bounds obtained by cutting plain approaches and their handling of the Branch-and-Bound tree has no competition, see (Rebennack et al., 2011; Rossi and Smriglio, 2001). There are several optimization solvers that include an implementation of the Branch-and-Cut algorithm which we use to evaluate the performance of our branching strategy. We considered the following two optimization solvers for this project: The Gurobi optimizer and the IBM ILOG CPLEX Optimizer. Both solvers provides the possibility of modifying the optimization process via callback functions. Gurobi is a new optimizer and is gaining a lot of popularity. It is also a very user friendly software and provide free academic licenses. However, as we found in the process of this work, the Gurobi optimizer does not provide the possibility of choosing the branch variable. This being a very important aspect to be able to implement our branching strategy, we turn to the *IBM ILOG CPLEX optimizer (version 12.6.3.0)*. This optimizer provides us with a Callback class which we use to choose our branching variable. The first is the *BranchCallback* class which we use to make a branch from the current node on the chosen branching variable. Finally, we also compare our optimization results with those of the branching strategy chosen by CPLEX optimizer for 13 different MIS problems.

3.2.2 Graph Objects

To be able create and handle graphs, we needed a graph class or module. Our preference immediately went to the Python graph module *networkx*. The version at the time of this project is *1.10*. Besides the graph object and graph routines in this module, we also make use of the Djikstra's shortest path algorithm implemented in the same module.

3.3 Parameters

The following parameters were also introduced to the optimization process.

VOC-L: This limits the number of violated odd cycles that were to be found by the algorithm. That is to say, whenever the total number of violated odd cycles found across the Branch Callbacks reaches *VOC-L* we stopped calling the odd cycle algorithm within subsequent callbacks. This meant that we hence branch on the feasible fractional variable with maximum degree in the graph **G**. Whenever this parameter is set to 0, it means the algorithm does not to attempt to find any violated odd cycles but instead the branching variable would be that with the maximum degree amongst the feasible variables. This was used to show that in some problems, where almost every attempt to find violated odd cycles failed, branching on the maximum degree variable resulted in the same number of nodes explored and only time would be gained by not searching for odd cycles. On the other hand, if the parameter were set to infinity,(∞), then at every node the algorithm attempted to find violated odd cycles. The table below shows the final upper bound, denoted (UB_{final}), the optimal value($IntOpt$), the time, and number of nodes explored by both branching rule use by CPLEX optimizer (left) and also that use by our branching rule. It also shows the different settings used for the parameter aforementioned and the number of violated odd cycles (VOC) found in each problem.

TABLE 3.1: Results of Branch-and-Cut Algorithm on 12 Problems

.	Default CPLEX Branching Rule				Branching on Violated Odd Cycles/Maximum degree				
Problems	UB_{final}	**IntOpt**	**Nodes**	**Time**	UB_{final}	**IntOpt**	**Nodes**	**Time**	**VOC/VOC-L**
brock200_1	22.0389	21	165620	1973.13	21.0000	21	109618	1311.38	0/0
brock200_1					21.0000	21	106099	7750	51454/∞
brock200_2	13.2582	12	3807	166.36	12.0000	12	1176	57.22	0/0
brock200_2					12.0000	12	1176	829.50	0/∞
brock200_3	16.3130	15	20129	369.13	15.0000	15	5330	147.20	0/0
brock200_3					15.0000	15	4681	2992.75	270/∞
brock200_4	17.7500	17	41471	531.03	17.0000	17	30373	769.16	0/0
brock200_4					17.0000	17	23155	10566.72	6000/∞
C125.9	36.4613	34	1220	5.25	34.0000	34	1733	13.24	0/0
C125.9					34.0000	34	2006	156.31	1008/∞
DSJC125.1	36.3889	34	1295	4.58	34.0000	34	3248	21.19	0/0
DSJC125.1					34.0000	34	3254	40.95	1628/∞
DSJC125.5	10.0000	10	254	7.08	10.0000	10	206	10.61	0/0
DSJC125.5					10.0000	10	206	162.31	1/∞
keller4	12.5000	11	3907	36.05	11.0000	11	2026	28.03	0/0
keller4					11.0000	11	2014	1218.81	57/∞
p_hat300-1	9.4195	8	1480	494.89	8.0000	8	706	492.72	0/0
p_hat300-1					8.0000	8	706	1633.81	0/∞
p_hat300-2	26.2193	25	850	81.95	25.0000	25	682	58.56	0/0
p_hat300-2					25.0000	25	548	508.70	80/∞
p_hat300-3	**	**	**	**	36.0000	36	95582	3520.31	0/0
p_hat300-3					36.0000	36	70371	15845.00	31409/∞
san200_0.7	19.2658	18	96815	1008.84	18.0000	18	45640	632.06	0/0
san200_0.7					18.0000	18	36492	10422.64	13085/∞
sanr200_0.9	43.0000	42	832427	4145.66	42.0000	42	448537	3146.50	0/0
sanr200_0.9					42.0000	42	333090	5727.50	166746/∞

** = could not be determine, computer ran out of memory

3.4 Analysis of the results

The violated odd cycle branching rule was better than the default strategy used by CPLEX in 10 out of 12 problems in terms of the number of nodes explored. Also, we are able to see from over 7 problems including brock200_1, brock200_3, brock200_4, and p_hat300-3, that branching strictly on the maximum degree variable was at least not better than branching on vertices on a violated odd cycle. The strength of our approach is strongly reflected in the sanr200_0.7 and sanr200_0.9 where our the number of nodes explored is significantly less than have the number of nodes by the default CPLEX algorithms. Also, the smaller number of violated odd cycles found in keller4 and p_hat300-2 is greatly reflected in the difference made by the rule compared to branching on maximum degree vertices. On the other hand, an extremely different result is observed for C125.9 where branching on maximum degree vertices or using our branching rule increases the number of nodes explored in the branch-and-cut tree. The properties of this graph should be further considered to be able to explain

this strange result. Moreover, the gap between the final upper bound, and the Optimal value of the objective function was zero only for 2 of 12 problems for the default strategy used by CPLEX. While for every problem, our branching rule, ended with a zero gap. The longer times taken can also be justified by the extra computations involve and the fact it provided a 100% certificate of optimality of the solution for all problems considered.

Chapter 4

Conclusion

4.1 Observations

The violated cycles detected in some of the graphs were all unexpectedly high in some graphs and low in other graphs. This had a significant effect on how the branching rule affected the brand-and-bound tree. And in some cases worsen the algorithm by increasing the number of nodes explored.

It is also evident from the results, that simply branching on the vertex with the maximum degree was not as good as branching on those vertices that belong to a violated odd cycles. The violated odd cycle branching rule was 10 of 12 times better than the rule use by CPLEX optimizer in terms of the number of nodes explored by the branch and bound tree. In the same line, 7 of 12 times better than branching simply on the maximum degree vertex.

To sum it all, branching on violated odd cycles proved to be a potentially excellent branching rule for maximum stable set integer programs.

4.2 Recommendations

In the violated odd cycle algorithm, we were only interested in finding any violated odd cycle. Defining a *most* violated odd cycle inequality as the odd cycle having minimum weight, we find it very likely that the following suggestions would improve the computational effectiveness of the violated odd cycle algorithm, see (Rebennack, Reinelt, and Pardalos, 2012).

- Instead of branching simply on a vertex in a violated odd cycle, it would be better to ensure that the cycle is the most violated odd cycle, or choose a violated odd cycle if the path weight was smaller than $\frac{1}{2} - \epsilon$, for some $\epsilon \geq 0$.
- Compute only odd cycles in G for a node, say v. That is, generate the tree or layered graph for at most one node of **G** as root node. Althought this is a heuristic, the speed up of the odd cycle algorithm could quite substantial. It also make the odd cycle found largely dependent on the root node.

4.3 Future works

Finally, one can already count one or two parts of this work that could be revisited and/or improved upon.

- Instead of using the violated odd cycles to choose a branching variable, it might be better to add this as a lazy constraint to the model, although this might lead to an exponential increase in constraints. Also, the algorithm used for finding violated odd cycles is not an exact algorithm, this makes the process less efficient.

- An exact algorithm for odd hole and antihole detection could rather be used on the subgraph induced by the fractional variables. From there, the resulting odd hole will contain the candidates for the branching variable. More on Odd hole and antihole detection is provided by Stavros and Leonidas, ((Nikolopoulos and Palios, 2004)).

- Also, it might have be better to restrict the search for violated odd cycles to some depth of the branch-and-bound tree. Unfortunately, one could not determine the depth of the tree from within the *Branch Callback* of the CPLEX Optimizer. So, an optimization solver that provides this functionality would be of great help for enhancing this branching rule.

- CPLEX Optimizer has over 5 built in branching rules and can be configured to use a particular rule to solve a given problem. We compared our results with the default branching rule chosen by CPLEX Optimizer for each problem. Indeed, the results does not clearly indicate how well our branching rule performs compared to a particular rule, but shows the default rule use by is not better than our proposed rule.

- Finally, another important property of a good algorithm is its speed. Due to possible variations that would arise from differences in implementation, we could not compare the running times required by the different branching rules. By doing this, one could also determine if the time needed to find violated odd cycles is justified by the number of nodes reduced in the branch-and-bound tree.

Appendix A

Flowchart of a Branch-and-Cut algorithm

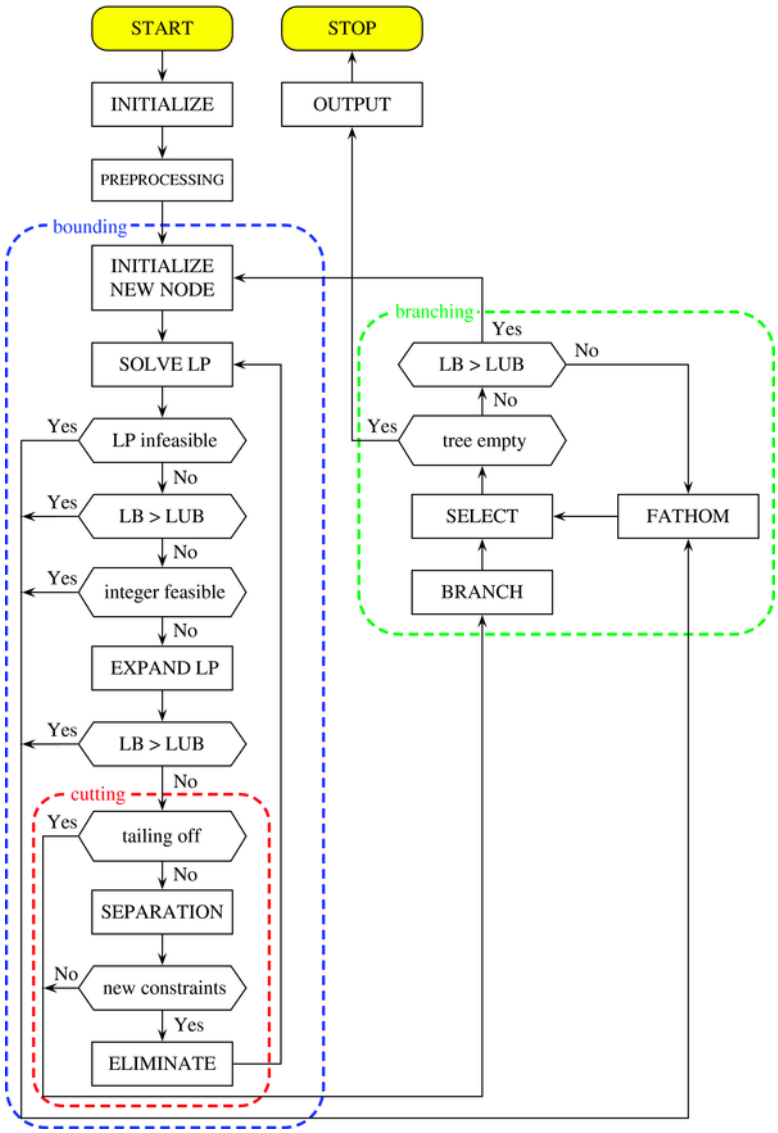

FIGURE A.1: A flowchart of the Branch-and-Cut Algorithm for maximization problem, see (Rebennack, Reinelt, and Pardalos, 2012)

Bibliography

Achterberg, Tobias, Thorsten Koch, and Alexander Martin (2005). "Branching rules revisited". In: *Operations Research Letters* 33.1, pp. 42–54. URL: http://linkinghub.elsevier.com/retrieve/pii/S0167637704000501.

Arora, S. and S. Safra (1992). "Probabilistic checking of proofs; a new characterization of NP". In: *Proceedings., 33rd Annual Symposium on Foundations of Computer Science* 45, pp. 2–13. URL: https://www.researchgate.net/publication/3513508_Probabilistic_Checking_of_Proofs_A_New_Characterization_of_NP.

Butenko, Sergiy (2003). "Maximum Independent set and related problems, with applications". In: *PhD thesis, University of Florida, USA, 2003.* URL: http://www.academia.edu/2617344/Maximum_independent_set_and_related_problems_with_applications.

Chudnovsky, Maria et al. (2006). "Strong Perfect Graph Theorem". In: *Annals of Mathematics* 164, pp. 51–229. URL: http://annals.math.princeton.edu/2006/164-1/p02.

E., Mitchell John (2002). "Branch-and-Cut Algorithms for Combinatorial Optimization Problems". In: *Handbook of Applied Optimization*, pp. 65–77.

Grotschel, Martin, Laszlo Lovasz, and Alexander Schrijver (1980). "Geometric Algorithms and Combinatorial Optimization". In:

Hawking, Stephen (2011). "A brief history of time, from the big bang to black holes". In:

Hoffman, Karla L. and Manfred Padberg (1993). "Solving Airline Crew Scheduling Problems by Branch-and-Cut". In: *Management Sciences* 39.6, pp. 1–26. URL: http://link.aip.org/link/?RSI/62/1/1.

Nemhauser, G.L. and G. Sigismondi (1992). "A strong cutting plane/Branch-and-Bound Algorithm for Node Packing". In: *Journal of Operational Research Society* 43.5, pp. 443–457. URL: link.springer.com/10.1057/jors.1992.71.

Nikolopoulos, Stavros D. and Leonidas Palios (2004). "Holes and Antihole detection in graphs". In: *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* 1.1, pp. 850–859. URL: http://link.springer.com/10.1007/s00453-006-1225-y.

Rebennack, Steffen (2008). "Stable Set Problem: Branch and Cut Algorithms". In: *Encyclopedia of Optimization Second Edition*, pp. 3676–3688.

Rebennack, Steffen, Gerhard Reinelt, and Panos M. Pardalos (2012). "A tutorial on branch and cut algorithms for the maximum stable set problem". In: *International Transactions in Operational Research* 19.5, pp. 161–199. URL: http://dx.doi.org/10.1111/j.1475-3995.2011.00805.x.

Rebennack, Steffen et al. (2011). "A Branch and Cut solver for the maximum stable set Problem". In: *Journal of Combinatorial Optimization* 4, pp. 434–457. URL: http://link.springer.com/10.1007/s10878-009-9264-3.

Robson, J.M (1986). "Algorithms for maximum independent sets". In: *Journal of Algorithms* 7.3, pp. 425–440. ISSN: 01966774. DOI: 10.1016/0196-6774(86)90032-5. URL: http://linkinghub.elsevier.com/retrieve/pii/0196677486900325.

Rossi, Fabrizo and Stefano Smriglio (2001). "Branch-and-cut algorithm for the Maximum Cardinality Stable Set Problem". In: *Operation Research Letters* 28.2, pp. 63–74. URL: http://linkinghub.elsevier.com/retrieve/pii/S0167637700000602.