Representation Changes in Combinatorial Problems: Pigeonhole Principle versus Linear Programming Relaxation *

Yury V. Smirnov and Manuela M. Veloso
School of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213-3891, USA

{smir, mmv}@cs.cmu.edu

Abstract

The Pigeonhole Principle (PHP) has been one of the most appealing methods of solving combinatorial optimization problems. Variations of the Pigeonhole Principle, sometimes called the "Hidden" Pigeonhole Principle (HPHP), are even more powerful and often produce the most elegant solutions to nontrivial problems. However, some Operations Research approaches, such as the Linear Programming Relaxation (LPR), are strong competitors to PHP and HPHP. They can also be applied to combinatorial optimization problems to derive upper bounds. It has been an open question whether PHP or LPR establish tighter upper bounds and how efficiently, when applied to the same problem. Challenged by this open question, we identify that the main reason for the lack of ability to compare the efficiency of PHP and LPR is the fact that different problem representations are required by the two methods.

We introduce a problem representation change into an Integer Programming form which allows for an alternative way of solving combinatorial problems. We also introduce several combinatorial optimization problems, and show how to perform representation changes to convert the original problems into the Integer Programming form. Using the new problem model, we re-define the Pigeonhole Principle as a method of solving Integer Programming problems, determine the difference between PHP and HPHP, prove that PHP has the same bounding power as LPR, and demonstrate that HPHP and Integer cuts are actually similar representation changes of the problem domains. The choice of representation is well-known to be one of the most difficult parameters to adjust and evaluate in machine learning algorithms. This work is therefore of particular relevance to machine learning researchers interested in understanding the impact of different representations. By analyzing representations used in the Operational Research area, we study and demonstrate the impact of representation choices in problem solving efficiency.

Keywords: Representation changes; Pigeonhole Principle; Integer Programming

*This research is sponsored in part by the National Science Foundation under grant number IRI-9502548. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations or the U.S. government.

1 Introduction

The Pigeonhole Principle (PHP) (Thucker, 1980) is traditionally considered the simplest and most elegant method of deriving tight upper bounds for a class of combinatorial optimization problems. In its original formulation, the Pigeonhole Principle (also known as the Dirichlet drawer principle) states that "it is impossible to place $N+1$ pigeons in N holes so that there is at most one pigeon in each hole." In this simple form, PHP looks like a naïve kindergarten-level rule. However, if taken to a higher level of different number of objects (pigeons) allowed in abstract units (holes), or used as a part of a multi-step logical proof, it quickly loses the nuance of obviousness. Though PHP has no clear-cut mathematical origin, its applications involve some of the most profound and difficult results in combinatorics of great relevance to Artificial Intelligence.

To solve a combinatorial optimization problem by PHP, one needs to represent a problem in such a way that the principle may be applied, i.e. to perform what we call the representation change. For example, to identify what in the problem should be mapped to objects (pigeons) and units (holes). For simple problems, the applications of PHP are almost straightforward and are obtained directly from the nature of the problems. However, often the proof by PHP requires additional heuristic knowledge that allows the effective representation change. If this representation change is found, then PHP easily produces a tight upper bound and an optimal solution. For some combinatorial optimization problems it is a very challenging task to find such a representation change. Hardness of this task encourages to look for alternative methods.

The introduction of polynomial-time algorithms solving Linear Programming problems allows some Operations Research methods to be applied also to combinatorial optimization problems to establish upper bounds for such problems. For example, the Linear Programming Relaxation (LPR) method has been applied effectively to solve some Integer Programming (IP) problems. LPR can be seen as requiring less effort to apply than PHP, because in general, unlike PHP, it does not need any additional knowledge or representation changes to provide upper bounds for IP problems.

The two approaches, namely PHP and LPR-based methods can be seen as "competitors" for solving combinatorial problems. It has been an open question[1] whether which method, if any, can provide better upper bounds and which method, if any, is more efficient, when applied to the same combinatorial optimization problems. This paper reports on our work in solving this open question. As hinted so far, our work analyzes carefully the problem representation issues involved in the two approaches. We formally introduce an appropriate representation change that makes the comparison and analysis possible.

Our work includes:

1. Re-defining PHP as a method of solving Integer Programming problems.

2. Proving that both PHP and LPR obtain the same upper bounds.

3. Providing an alternative approach to solving combinatorial optimization problems for which the effective representation change required by PHP is hard to find. The al-

[1]Identified at the First International Workshop on AI and OR, Portland, OR, June, 1995.

ternartive approach that can be implemented in AI systems, consists of the following steps:

(a) Convert a combinatorial optimization problem into an Integer Programming form.

(b) Apply LPR to obtain an upper bound B which, as we proved, is the same as the upper bound obtained by PHP.

(c) Construct an optimal solution of value B.

In this work, we also determine the difference between the original and the "Hidden" Pigeonhole Principles, analyze the latter one and compare its bounding power with that of Integer cuts.

2 Our Representation Change Approach

The Pigeonhole Principle is the most simpleminded ideas imaginable, yet its generalizations involve some of the most difficult results in combinatorics(Thucker, 1980). A more sophisticated fashion of applying PHP is sometimes called the "Hidden" Pigeonhole Principle. In such cases, an optimal solution is usually nontrivial and the application of the Hidden Pigeonhole Principle (HPHP) is probably the simplest way of proving the given solution's optimality. Though the difference between PHP and HPHP has never been clearly determined, the transition phase in the difficulty of proving the solution's optimality lies exactly in a possibility of applying PHP vs. HPHP.

Linear Programming Relaxation (LPR) is one of the most popular Operations Research methods of establishing upper bounds for Integer Programming problems(Nemhauser and Wolsey, 1985). Its main calculation engine is based on the discovery of polynomial methods of solving Linear Programming problems, such as the Ellipsoid algorithm(Khachian, 1979). Actually, LPR is a two-step procedure consisting of relaxing integer requirements and solving the corresponding Linear Programming problem.

Simplisticly, LPR can be viewed as a black-box with a particular instance of an Integer Programming problem as an input and a calculated upper bound as an output. From this point of view, LPR looks preferable to PHP, because it does not need any heuristic knowledge in deriving upper bounds. LPR can be applied to any instance of an Integer Programming problem without additional representation changes. Thus, the main questions of the competitive analysis between LPR and PHP can be stated as the *knowledge representation problem*: Which form of presenting combinatorial optimization problems allows to establish tighter bounds?

In the following sections, we determine the difference between PHP and HPHP, compare computational power of PHP and LPR, overview similarities between HPHP, PHP with heuristics, and Integer cuts for Integer Programming problems. We illustrate the discussion by a series of combinatorial optimization problems of a gradually increasing complexity.

2.1 Converting Problems into the Integer Programming Form

We identified that the main reason of the lack of ability to compare the efficiency of PHP and LPR is that different problem representations are required by the two methods. Our work consisted of developing appropriate changes of representations that made possible a competitive analysis of the efficiency of these two methods.

To make both PHP and LPR applicable to the same combinatorial optimization problems, we perform representation changes for each problem discussed in this paper to convert them into Integer Programming problems of the following form:

$$\begin{aligned}
\text{goal function :} \quad & max(\sum_{i=1}^{N} c_i x_i) \\
\text{constraint set } J: \quad & \sum_{i=1}^{N} a_{ij} x_i \leq b_j \quad j = 1, \ldots, M \\
\text{integrality requirements :} \quad & x_i \in Z \quad i = 1, \ldots, N
\end{aligned}$$

A combinatorial optimization problem is defined as finding a solution $x^* = (x_1^*, \ldots, x_N^*)$ that is feasible, i.e. satisfies all the constraints from J and the integrality requirements, and also attains the best value of the goal function among all feasible solutions.

We considered a series of known combinatorial optimization problems, to which the Pigeonhole Principle is applicable. The empirical evidence obtained through this study suggests that the following definition reflects correctly the combinatorial nature of the Pigeonhole Principle. We show the correctness of this definition through a set of problems that illustrate applications of PHP and LPR.

> **Definition:** *Suppose that an Integer Programming problem of the above type has a known feasible solution x^*. We say that the problem admits a proof by the Pigeonhole Principle if there exists a subset of constraints $\hat{J} \subseteq J$, such that the sum of the left-hand sides of the inequalities from $\hat{J}$ (repetitions are allowed in $\hat{J}$) is a multiple of the goal function*
>
> $$\sum_{j \in \hat{J}} \sum_{i=1}^{N} a_{ij} x_i = k \sum_{i=1}^{N} c_i x_i$$
>
> *and the value of x^* equals the same multiple of the sum of the right-hand side constants*
>
> $$\sum_{j \in \hat{J}} b_j = k \sum_{i=1}^{N} c_i x_i^* \, .$$
>
> *If such a solution x^* is not known, then the best value*
>
> $$\min_{\hat{J} \subseteq J} \frac{1}{k} \sum_{j \in \hat{J}} b_j \ , \quad \text{such that} \ \sum_{j \in \hat{J}} \sum_{i=1}^{N} a_{ij} x_i = k \sum_{i=1}^{N} c_i x_i$$
>
> *holds for some positive $k > 0$, is called an upper bound obtained by PHP.*

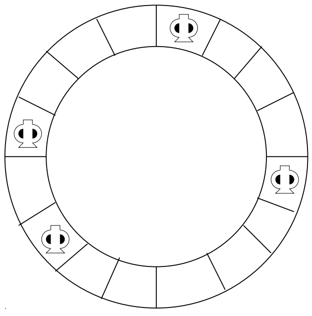
In general, the procedure of considering a subset of constraints and summing them up (possibly with positive weights) results in setting upper bounds for the value of an optimal solution. Any feasible solution provides a lower bound for the optimal value of the problem. We identify applications of PHP with finding a subset of constraints and deriving the optimal value of the goal function. In this case, the established upper bound matches the lower bound provided by the found solution.

The introduced representation change allows to re-formulate the original PHP statement with 11 pigeons and 10 holes as the following IP problem:

$$max(\sum_{i=1}^{11}\sum_{j=1}^{10} x_{ij})$$
$$\sum_{i=1}^{11} x_{ij} \leq 1 \quad j = 1, \ldots, 10$$
$$x_{ij} \in \{0,1\},$$

where x_{ij} represents the amount of the ith pigeon in the jth hole. Thus, if we drop integer requirements and sum up all the constraints, we obtain an upper bound: $\sum_{i=1}^{11}\sum_{j=1}^{10} x_{ij} = \sum_{j=1}^{10}\sum_{i=1}^{11} x_{ij} \leq 10$. Since an obvious solution $\{x_{ii} = 1 \text{ for } i = 1, \ldots, 10;\ x_{ij} = 0 \text{ for } i \neq j\}$ provides the same value, we proved it to be optimal. For this simple problem, PHP and IRP approaches are identical.

In most obvious cases, the desired subset of constraints is the set of original inequalities. Consider, for example, the problem of placing chess kings on a circular chessboard with even number of squares (see Figure 1). One is supposed to find the maximal number of kings that can be placed on such a board so that no king is attacking another. Recall that a chess king attacks all its adjacent squares. For the board presented in Figure 1, we get the following IP problem:



$$max(\sum_{i=1}^{N} x_i)$$
$$x_1 + x_2 \leq 1$$
$$x_2 + x_3 \leq 1$$
$$\ldots$$
$$x_{N-1} + x_N \leq 1$$
$$x_N + x_1 \leq 1$$
$$x_i \in \{0,1\}$$

Figure 1: Chess kings on a circular chessboard with N positions and corresponding IP representation. Kings cannot be in adjacent positions, i.e. attack each other.

If we sum up all the inequalities, we get $2\sum_{i=1}^{N} x_i \leq N$. On the other hand, the same upper bound can be derived by applying a combination of two-coloring and PHP: if we color the circular board with even number of squares in alternating black and white colors, we can place kings on either color attaining the optimum value of $N/2$ derived this way by PHP.

The case of summing up a proper subset of constraints is more complicated. One has to come up with a rule (heuristic) of finding a desired subset of constraints. For some problems, it is easy to find such a subset, whereas for others it is not obvious. For example, the popular N-Queen problem(Hoffman et al., 1969; Nauck, 1850), which is the problem of placing the maximal number of chess queens on NxN-chessboard, so that no queen is attacking another, can be represented as the following IP problem:

$$max(\sum_{i=1}^{N}\sum_{j=1}^{N} x_{ij})$$

$$\text{for each } i \in \{1,\ldots,N\} \text{ and } j \in \{1,\ldots,N\} \quad \begin{cases} \sum_{k=1}^{N} x_{ik} \leq 1 \\ \sum_{k=1}^{N} x_{kj} \leq 1 \\ \sum_{(k,l)\in I(i,j)} x_{kl} \leq 1 \\ \sum_{(k,l)\in J(i,j)} x_{kl} \leq 1 \\ x_{ij} \in \{0,1\} \end{cases}$$

where $I(i,j)$ and $J(i,j)$ are the sets of squares which a chess queen threatens along two diagonals from the square (i,j), including the square (i,j). In this form we have four groups of constraints: row, column, and two diagonal constraints, one of each type per square. For this problem it is easy to find a subset of constraints that provides the tight upper bound. If we sum up N inequalities corresponding only to row constraints for squares from different rows, we get N as the upper bound: $\sum_{i=1}^{N}\sum_{j=1}^{N} x_{ij} \leq N$. Beginning with $N = 4$, the problem has an N-Queen solution(Hoffman et al., 1969) (see Figure 2).

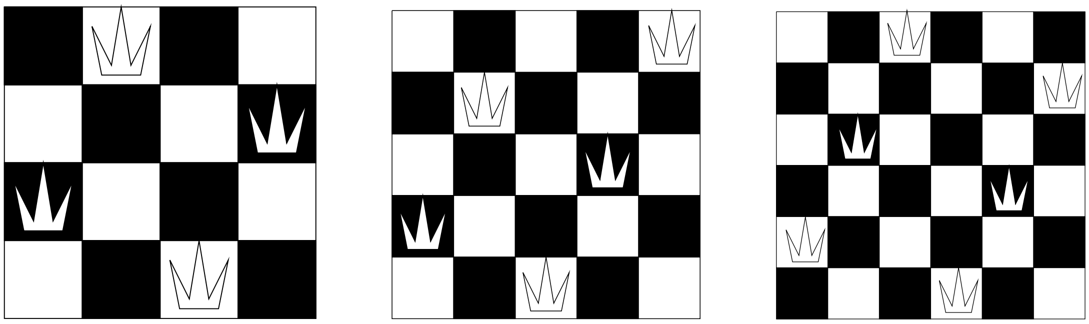


Figure 2: Solutions of the N-Queen problem for N = 4, 5, 6.

The Chess Knight problem(Bennett and Potts, 1967) is one of the jewels of PHP applications; it is the one for which the selection of a constraint subset is a challenging task. We discuss this problem in the following section in detail.

3 PHP vs. LPR

In this section, we discuss relations between the original Pigeonhole Principle and the Linear Programming Relaxation. For some problems, it is tempting to apply LPR in a brute-force manner. Instead of deriving an additional heuristic rule, which will tell how to select the desired subset of constraints, one can apply already developed methods of Linear Programming with the hope of deriving the same or even better bound. This brings us back to the main question of this paper: Is it true that PHP and LPR always provide the same upper bound? The following theorem answers this question:

Theorem 1 *If an Integer Programming (IP) problem admits a proof by the Pigeonhole Principle, then LPR provides the same optimal value. Conversely, a bound derived by LPR can also be obtained by PHP.*

Proof: If an IP problem admits PHP in deriving the tight upper bound, then there exists a feasible integer solution x^* attaining the derived bound. On the other hand, there exists a subset of constraints $\hat{J}$, sum of the left-hand sides of which is an integer multiple of the goal function: $\sum_{j\in\hat{J}}\sum_i a_{ij}x_i = k\sum_i c_i x_i$, and the value of x^* is the same multiple of bounding constants: $\sum_{j\in\hat{J}} b_j = k\sum_i c_i x_i^*$.

If we apply LPR to this IP problem, it will provide an integer or a fractional optimal solution x^{**} for a relaxed problem. Since PHP has derived the tight upper bound, and the relaxed problem is obtained from the original IP problem by dropping integrality requirements, the value of LPR's solution x^{**} is at least as large as PHP's tight upper bound: $\sum_{i=1} c_i x_i^{**} \geq \sum_{i=1} c_i x_i^*$. According to Linear Programming theory, there exists a subset of the constraint set, called "active" constraints, such that the goal function is a positive weighted sum of the left-hand sides of the constraints from this subset, and the optimal solution has no slack for each of the constraints, that is, satisfies them as equality. Since all the coefficients in the constraints J and the goal function are integer, all the weights (positive coefficients) of the weighted sum are rational. We can scale them up to make all of them integer.

Thus, the set of "active" constraints forms the desired subset $\hat{J}$ and positive integer weights determine the number of repetitions in $\hat{J}$. Therefore, the value of the solution derived by LPR cannot exceed the upper bound provided by PHP and, conversely, it can be mimicked by PHP to establish the upper bound. ∎

Theorem 1 provides the following answer to the main question of the paper: *If the optimal value is obtained by PHP for a combinatorial optimization problem stated in the Integer Programming form, LPR provides the same bound as PHP.* We demonstrate that this result is nontrivial by solving the Chess Knight problem(Bennett and Potts, 1967): "What is the maximal number of chess knights that can be placed on the 8x8 chessboard in such a way that they do not attack each other?"

The classical elegant solution relies on the existence of a Hamiltonian tour of length 64 following the knight moves. One of such tours is presented in Figure 3.

One can split the tour into 32 pairs of chess squares that are adjacent in the sense of the knight move, and apply PHP: each pair can contain at most one knight, otherwise two knights

8	43	24	53	6	41	22	51
25	54	7	42	23	52	5	40
44	9	62	15	46	31	50	21
55	26	45	32	63	14	39	4
10	33	16	61	30	47	20	49
27	56	29	64	13	60	3	38
34	11	58	17	36	1	48	19
57	28	35	12	59	18	37	2

Figure 3: Hamiltonian tour on a chessboard for the knight.

would attack each other. For example, consider pairs of squares $(1,2)$, $(3,4)$,...,$(63,64)$. None of them can accommodate more than one knight (see Figures 3 and 4). Thus, 32 pairs of adjacent squares can accommodate at most 32 knights. Although this simple proof does not provide us with a solution to the problem, (for example, it allows to place knights on squares 4 and 5), it gives an upper bound.

Moreover, this proof gives us an impression of using a "hidden" application of the Pigeonhole Principle, whereas the Hamiltonian tour is just a heuristic for finding a subset of the constraint set. The Chess Knight problem for the standard chessboard can be presented as the following Integer Programming problem:

$$max(\sum_{i=1}^{8}\sum_{j=1}^{8} x_{ij})$$
$$x_{ij} + x_{kl} \leq 1 \quad i = 1, \ldots, 8 \;\; j = 1, \ldots, 8 \;\; (k,l) \in U(i,j)$$
$$x_{ij} \in \{0, 1\}$$

where x_{ij} represents the amount of knights in the square (i,j); $U(i,j)$ is the set of squares on the chessboard which a chess knight threatens from the cell (i,j) (see Figure 4).

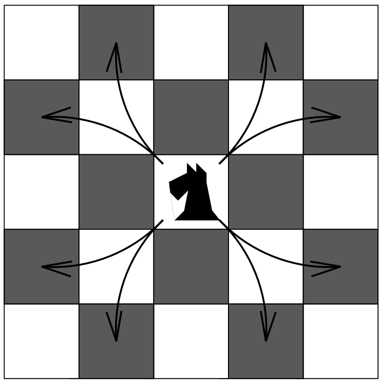


Figure 4: The set of adjacent squares for the knight on the chessboard.

If applied to the Chess Knight problem, the Linear Programming Relaxation method provides the upper bound of 32. Unlike PHP, LPR does not require heuristics to identify any subset of constrains, its calculational routine considers "active" constraints within solving process. However, LPR is likely to produce fractional solutions, say $x_i = 1/2$ for $i = 1, 2, \ldots, 64$. Theorem 1 shows that, if applied, the original Pigeonhole Principle will provide the same value. So, if the Hamiltonian tour heuristic were not known, the application of LPR would suggest 32 as the best upper bound that can be obtained by the original PHP. Since a chess knight alternate colors (see Figure 4), one can place 32 knights on the chess squares of the same color, thus obtaining the optimal value and the solution for the Chess Knight problem.

We showed that, if an optimal solution of value B for a combinatorial optimization problem is derived by the Pigeonhole Principle, an optimal solution for the last problem in the following chain of representation changes has the same value B:

Original combinatorial problem$\rightarrow$ IP problem $\rightarrow$ Linear Programming problem.

Unfortunately, LPR often outputs a fractional optimal solution, which does not give any hint on how to transform it into an integer one of the same value. On the other hand, applications of PHP often lead to an optimal integer solution.

4 More Complicated Applications of PHP

In the previous sections, we were able to apply PHP in a brute-force manner, because each Integer Programming problem contained a subset of constraints, the sum of which provided the desired bounds for values of the goal functions. We call such a case a regular application of PHP, as opposed to the "Hidden" Pigeonhole Principle (HPHP) that requires additional representation changes and heuristic knowledge to fulfill a similar task.

This section is devoted to the discussion of HPHP and its relation to the original Pigeonhole Principle. To make it more intuitive, we illustrate the discussion by the Mutilated Checkerboard problem(Newell, 1965):

> **Mutilated Checkerboard Problem:** Consider an NxN checkerboard with two opposite corners removed (see Figure 5). Can one cover this "mutilated" checkerboard by non-overlapping domino pieces, each covering two squares of the checkerboard?

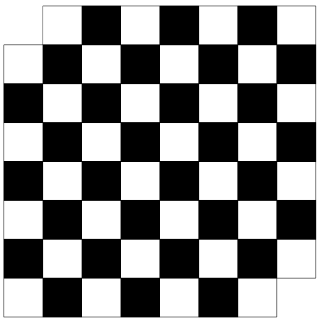


Figure 5: A mutilated checkerboard.

Following the proposed approach, the first necessary step is the representation change converting this problem into an Integer Programming problem. To accomplish this step, one needs to model the fact that domino pieces do not overlap in the Integer Programming form. We assign variables x^v_{ij} to vertical splitting lines, where i is the row number and j is the splitting line number in ith row between squares (i,j) and $(i,j+1)$. In the same way, we define variables x^h_{ij} for horizontal splits between squares (i,j) and $(i+1,j)$. Variables x^v_{ij} and x^h_{kl} denote the location of the middle line of domino pieces at ith vertical splitting line, jth row, and kth horizontal splitting line, lth column, respectively. In these terms, non-overlapping is represented by the following set of constraints:

$$x^v_{ij} + x^v_{i-1j} \leq 1 \quad x^v_{ij} + x^v_{i+1j} \leq 1 \quad x^v_{ij} + x^h_{i-1j-1} \leq 1 \tag{1}$$

$$x^v_{ij} + x^h_{i-1j} \leq 1 \quad x^v_{ij} + x^h_{ij-1} \leq 1 \quad x^v_{ij} + x^h_{ij} \leq 1 \tag{2}$$

Figure 6 demonstrates a part of a checkerboard and the correspondence of the variables to the splits.

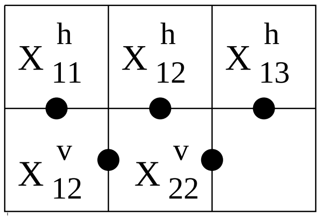


Figure 6: Modeling domino overlap as a set of inequalities.

The goal function is the sum of the domino-piece variables: $\sum_{ij} x^v_{ij} + \sum_{ij} x^h_{ij}$. It is easy to guess a fractional optimal solution produced by LPR: $x^*_{**} = 1/2$. Unfortunately, it does not help us to find an optimal integer solution. If the original checkerboard has the size NxN with an odd N, a simple PHP application shows that one can use at most $\frac{N^2-3}{2}$ domino pieces for the non-overlapping covering of the mutilated checkerboard: each piece contains two squares, and there are $N^2 - 2$ squares in the mutilated checkerboard. Therefore, one can put at most $\lfloor \frac{N^2-2}{2} \rfloor = \frac{N^2-3}{2}$ non-overlapping domino pieces. Actual attempts to cover the mutilated checkerboard with odd sizes readily give an optimal solution.

If N is even, simple PHP does not put any additional restrictions on the number of pieces. However, none of the attempts to cover the mutilated checkerboard by domino pieces achieves the desired bound of $\frac{N^2-2}{2}$, all constructed solutions provide at most $\frac{N^2-4}{2}$ pieces. Thus, neither PHP nor IRP provide a tight upper bound. Nonetheless, the heuristic of two-coloring of the mutilated checkerboard and applying PHP to the monochrome set of squares of smaller size completes the proof of the fact that $\frac{N^2-4}{2}$ is actually the upper bound for covering the mutilated checkerboard with even sizes. If we color the original checkerboard in usual black-and-white chess colors (see Figure 5) and then cut off 2 opposite corner squares (of the same color), the remaining mutilated checkerboard contains unevenly colored square sets. The mutilated checkerboard presented in Figure 5, for example, has 30 black squares and 32 white squares. Since each domino piece contains 2 squares of the opposite colors, applying PHP to the smaller set of black or white squares, we obtain the upper bound of $\frac{N^2-4}{2}$.

Theory of Integer Programming suggests to apply Integer cuts, for example, Chvatal-Gomory's cuts, to extend the set of valid inequalities. The main idea of the Chvatal-Gomory's cut (CG-cut) is to use the integrality of solutions. If a weighted sum of the inequalities derived so far contains the left-hand side with integer coefficients, the right-hand side can be rounded down to the nearest integer: $\sum_{i=1}^{N} a_i x_i \leq b$ implies $\sum_{i=1}^{N} a_i x_i \leq \lfloor b \rfloor$, for integer $a_i,\ x_i\ \ i = 1, \ldots, N$. In a certain sense CG-cuts look as simple as the original Pigeonhole Principle. However, CG-cuts allow to derive constraints that cannot be deduced from the initial set of inequalities by taking weighted sums.

We prove the correctness of the Integer Programming formulation of the Mutilated Checkerboard problem and derive an upper bound for it by means of Chvatal-Gomory's cuts and the Hidden Pigeonhole Principle. We model chess-board as an undirected graph $G = (V, E)$, which is actually a square lattice: vertices correspond to the centers of the checkerboard squares, undirected edges connect adjacent squares. If we consider 0/1-weights for all edges, a placement of a domino piece corresponds to the weight of 1 assigned to the correspondent edge. The set of inequalities introduced in this section restricts the weights assigned to adjacent edges, two domino pieces do not overlap if and only if the correspondent edges (with 1-assignment) are not adjacent.

Since we are about to apply the technique of Integer cuts to the Mutilated Checkerboard problem, we demonstrate a reduction from the set of pairwise constraints to the Exclusive Rule for a clique K_N, where a clique is a complete graph with $N \geq 2$ vertices.

Lemma 1 *If a clique K_N admits exclusively 0/1-assignments to its vertices $v_1, \ldots, v_N$ and, for each pair of vertices (v_i, v_j), at most one vertex can be assigned to 1, then there is at most one vertex assigned to 1 in the whole clique K_N.*

Proof: There are $\frac{N(N-1)}{2}$ inequalities of the type $x_i + x_j \leq 1$. We would like to prove that this collection of constraints implies a single clique-inequality $\sum_{i=1}^{N} x_i \leq 1$ (Exclusive Rule) for 0/1-variables $x_i\ \ i = 1, \ldots, N$. We prove it by induction on the number of vertices.
Induction Base: If $N = 2$, the given inequality and the clique-inequality coincide.
Induction Step: Suppose that we can derive all N clique-inequalities for each of the N sub-cliques of size $N-1$. If we sum them up, we get:

$$\sum_{j=1}^{N} \sum_{i \neq j} x_i \leq \sum_{j=1}^{N} 1,$$

which implies that $(N-1) \sum x_i \leq N$ or, equivalently, $\sum x_i \leq N/(N-1)$. Since $N > 2$, $\lfloor \frac{N}{N-1} \rfloor = \lfloor 1 + \frac{1}{N-1} \rfloor = 1$. If we apply CG-cut to the last inequality, we get the desired N-clique-inequality: $\sum_{i=1}^{N} x_i \leq 1$ ∎

Four (or less) splits that form the border of the square (i, j) correspond to four variables that model the placement of domino pieces. According to Lemma 1, if we consider all six pairwise inequalities (1-2) involving these four variables, CG-cuts provide the Exclusive Rule for the square. This Exclusive Rule corresponds to the requirement of non-overlapping of domino pieces over the square (i, j). The addition of the clique-inequalities derived by CG-cuts to the initial set of constraints constitutes a representation change of the IP problem.

However, if we apply LPR to the new problem with the extended set of constraints, we obtain an improved upper bound of $\frac{N^2-4}{2}$. The same bound can be derived using HPHP, by considering a monochromatic set of squares of smaller size.

The proof that uses HPHP, relies on the additional knowledge that each domino piece covers a bi-chromatic configuration, whereas the combination of LPR with CG-cuts takes into account this argument automatically. If the two-coloring heuristic were not known, then LPR with CG-cuts could be used to obtain the tight upper bound of $\frac{N^2-4}{2}$. After that, it is relatively easy to construct a solution attaining this value, which is proven to be optimal. However, HPHP's proof constructively leads to an optimal solution, whereas LPR with CG-cuts are likely to produce a fractional solution with no hint on how to construct an integer one. The relations between HPHP and LPR with Integer cuts are very similar to those between PHP and LPR.

5 Conclusions

In this paper, we showed how through an effective representation change we solved the problem of comparing the bounding power of the Pigeonhole Principle and the Linear Programming Relaxation method. We proved that both methods establish the same upper bounds, when applied to the same problems. The Pigeonhole Principle is more intuitive and often shows whether the upper bound is tight, whereas Linear Programming Relaxation can be applied to any instance of an Integer Programming problem without the use of additional knowledge or representation changes.

We illustrated our new representation change in several combinatorial optimization problems, and demonstrated how this representation change can be utilized to provide an alternative way of solving such problems, which is easier to implement in AI planning systems than PHP or HPHP.

References

Bennett, B. and Potts, R. (1967). Arrays and brooks. *Journal for the Australian Mathematical Society*, pages 23–31.

Hoffman, E., Loessi, J., and Moore, R. (1969). Constructions for the solution of the m queens problem. *National Mathematics Magazine*, March-April:66–72.

Khachian, L. (1979). A polynomial algorithm in linear programming. *Soviet Math. Dokaldy*, 20(1).

Nauck (1850). Schach. *Illustrierte Zeitung*, 361:352.

Nemhauser, G. and Wolsey, L. (1985). *Integer and Combinatorial Optimization*. John Wiley & Sons.

Newell, A. (1965). Limitations of the current stock of ideas about problem solving. In Kent, A. and Tualbee, O., editors, *Electronic Information Handling*. Spartan Books, Washington, DC.

Thucker, A. (1980). *Applied Combinatorics*. John Wiley & Sons.