EXPLOITING SYMMETRY TO IMPROVE THE PERFORMANCE OF BRANCH-AND-BOUND FOR FINDING MAXIMUM INDEPENDENT SETS

by

Paul Richard Hahn

Submitted in Partial Fulfillment

of the Requirements for the Degree of

Master of Science in Mathematics

with Operations Research and Statistics Option

New Mexico Institute of Mining and Technology

Socorro, New Mexico

May 21, 2007

ABSTRACT

We present an algorithm that exploits symmetry in graphs to improve search for maximum independent sets. In particular, we expedite search on graphs arising from coding theory by search tree pruning in a branch and bound algorithm.

In the context of coding theory, graphs are constructed relative to some pre-specified error such that independent sets correspond to error-correcting codes. Accordingly, optimal codes correspond exactly to maximum independent sets. We present computational results for two specific types of coding errors. In each case we are able to demonstrate for the first time that the best known solution is in fact the best possible solution.

We also extend our method to arbitrary graphs by using a fast heuristic program for identifying graph automorphisms. This combination of symmetry breaking and heuristic symmetry identification represents advantageous cooperation between exhaustive search and a heuristic method.

ACKNOWLEDGMENT

I would like to thank my advisor Brian Borchers who has tried his hardest to make some kind of mathematician out of me.

Similarly, I must thank Oleg Makhnin for his valuable instruction and good humor.

I express my gratitude to Drs. Bill Stone and Bixiang Wang for sitting on my thesis committee.

I would like to thank Danielle and Christian Lucero for adopting me here in Socorro and making sure I was well taken care of.

Likewise I thank Norelle "Norul" Shlanta for keeping me fed and making sure I didn't procrastinate too badly with my course work.

Also thanks to Aaron Wilson for being willing to argue with me, although I suspect he couldn't not have.

Most of all, thanks to my partner Shira Katseff, for everything.

This thesis was typeset with LaTeX[1] by the author.

[1] LaTeX document preparation system was developed by Leslie Lamport as a special version of Donald Knuth's TeX program for computer typesetting. TeX is a trademark of the American Mathematical Society. The LaTeX macro package for the New Mexico Institute of Mining and Technology thesis format was adapted from Gerald Arnold's modification of the LaTeX macro package for The University of Texas at Austin by Khe-Sing The.

TABLE OF CONTENTS

LIST OF FIGURES **iv**

1. PRELIMINARIES **1**

1.1 Introduction 1

1.2 Definitions From Graph Theory 1

1.3 Branch-and-Bound for MIS 3

1.4 The Lovasz ϑ Bound 10

1.5 Error-correcting Codes 11

1.6 Augmenting Branch-and-Bound with Additional Information . . 17

2. EXPLOITING SYMMETRY TO SOLVE SLOANE'S MIS CHALLENGE GRAPHS **20**

2.1 Introduction 20

2.2 Symmetry in 1et and 1tc Error Graphs 24

2.3 Computational Results 27

3. APPLICATIONS TO ARBITRARY GRAPHS **33**

3.1 Finding Symmetries 33

4. Conclusions **39**

Bibliography **41**

LIST OF FIGURES

1.1 Example of an undirected graph. 4

1.2 Branch-and-Bound binary search tree 5

1.3 Graph for branch-and-bound example. 7

1.4 Branch-and-Bound Example 9

1.5 The 1tc confusion graph for $n = 4$. 15

1.6 The 1et confusion graph for $n = 4$. 16

1.7 Checking for nodes with neighborhoods that are cliques. 19

2.1 Branch-and-bound example graph revisted. 21

2.2 Example graph revisited to illustrate orbits and symmetry. . . . 22

2.3 Branch-and-Bound Example With Symmetry Breaking and Constraint Propagation 24

2.4 Factor of Savings From Identifying Symmetry 30

2.5 Computational Results for 1tc.2048 and 1et.2048 31

This thesis is accepted on behalf of the faculty of the Institute by the following committee:

Brian Borchers, Advisor

Paul Richard Hahn   Date

CHAPTER 1

PRELIMINARIES

1.1 Introduction

The principles behind the symmetry-exploiting branch-and-bound algorithm presented in this thesis are quite straightforward. That said, fully appreciating the details of the algorithm – what it achieves and how – involves bringing together elements from various corners of mathematics that many readers will not have had occasion to visit recently. With this in mind, we begin with a not insignificant "preliminaries" section devoted to bringing the reader up to speed on the ideas and notation that will be necessary in explaining what new contributions the present manuscript offers. Subsequent sections are then devoted to detailing these contributions.

1.2 Definitions From Graph Theory

In what follows, the combinatorial objects of primary interest will be *undirected graphs*. Recall that an undirected graph is given by the ordered pair $(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a finite set of *vertices* and $\mathcal{E}$ is a set of *edges* consisting of two-element subsets of $\mathcal{V}$[7]. In what follows we will denote an edge using the notation (x, y) to mean $\{x, y\} \subseteq \mathcal{E}$.

Relative to some graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ we have the following definitions.

Definition 1.1. *An* **independent set** *is a subset* $I \subseteq \mathcal{V}$ *such that* $(x, y) \notin \mathcal{E}$

for every $x, y \in I$.

Definition 1.2. *A* **maximal independent set** *is an independent set I such that for every $x \in \mathcal{V} - I$, $I \cup \{x\}$ is not an independent set. In other words, a maximal independent set is an independent set that cannot be enlarged by adding another vertex from $\mathcal{V}$.*

Definition 1.3. *A* **maximum independent set** *(MIS) is an independent set M such that $|M| \geq |I|$ for every independent set I on $\mathcal{G}$, where $|\cdot|$ denotes set cardinality.*

We take particular note of two facts. First, for any given graph there may be many maximal independent sets with cardinality strictly less than that of a maximum independent set. Second, there may also be more than one maximum independent set, though by definition all will have the same cardinality. Indeed, these two facts contribute to the difficulty of finding maximum independent sets.

Formally we can characterize the difficulty of this problem in the language of *computational complexity* as discussed in [4, 11]. In particular we can look at the *recognition problem* associated with finding maximum independent sets:

> Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and an integer k, does $\mathcal{G}$ have an independent set of size k?

For arbitrary k and $\mathcal{G}$ this problem has no known *polynomial-time* algorithm, an algorithm which runs in $O(|\mathcal{V}|^c)$ time for some fixed c. Exhaustive search would

of course yield our answer, but such a process clearly takes time exponential in the number of vertices. In fact, this problem is known to be NP-*complete*, meaning that there can only be a polynomial time algorithm for its solution in the unlikely event that $P = NP$ [4, 11]. Of course, this is one of the outstanding problems of contemporary mathematics. For all practical purposes, the MIS recognition problem takes a great deal of time to solve with certainty.

This caveat "with certainty" bears emphasis. For, on the one hand, there are fast procedures or *heuristics* for finding large maximal independent sets [6]. On the other hand, while oftentimes it turns out that these sets are also maximum independent sets, proving the matter requires significantly more time.

In the next section we consider a class of algorithms that tries to capitalize on the efficiency of heuristics in an effort to speed up the process of exhaustive search.

1.3 Branch-and-Bound for MIS

Consider some fixed enumeration of $\mathcal{V}$, $(v_1, v_2, v_3, ...v_m)$, where $m = |\mathcal{V}|$. We define an *assignment* on this ordered set as follows.

Definition 1.4. *Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, an* **assignment** *is a partition of $\mathcal{V}$ into two non-intersecting subsets, P_0 and P_1.*

To each such assignment we may then correspond a bit string of length m, denoting set membership by 1's and 0's. We thus create a string $x = x_1x_2x_3...x_m$ with the property that $x_i = 0$ if $v_i \in P_0$ and $x_i = 1$ if $v_i \in P_1$. This correspondence makes clear the exponential nature of our problem and provides a simple

way to think about exhaustive search for maximum independent sets: we just enumerate all possible x's and check for those corresponding to the cases where P_1 is an independent set.

Next, we consider a search tree over all *partial assignments*.

Definition 1.5. *Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a* **partial assignment** *is a partition of $\mathcal{V}$ into three non-intersecting subsets, P_0 and P_1 and U (for "unfixed").*

Such a tree has height m, with $2^{m+1} - 1$ total nodes [1], and has a last row containing all 2^m assignments on $\mathcal{V}$. In the following diagrams we provide a simple example graph and the corresponding partial assignment binary tree.

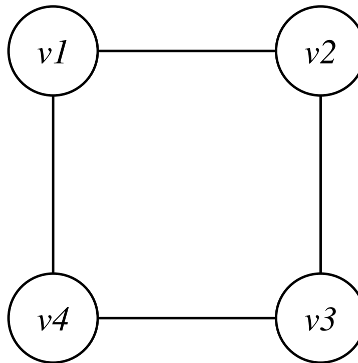


Figure 1.1: Example of an undirected graph.

[1] The present manuscript adopts the convention that vertices of a branch and bound search tree will be referred to as "nodes" to distinguish them from vertices of an arbitrary graph $\mathcal{G}$.

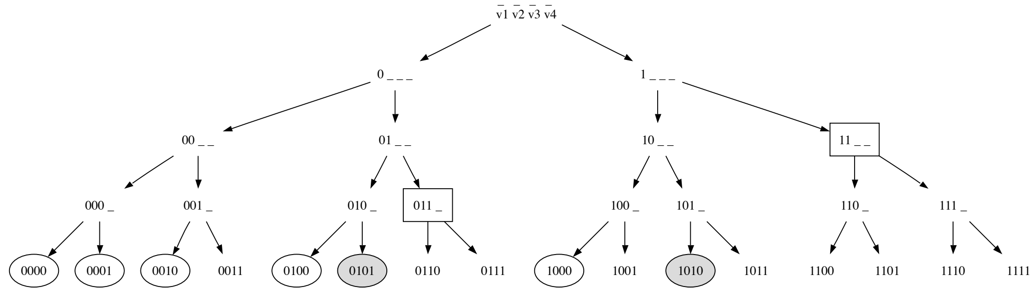


Figure 1.2: Branch-and-Bound binary search tree

Full assignments which correspond to independent sets are noted as ovals; filled ovals indicate those of maximum size. Boxed nodes represent partial assignments whose '1' elements already are not independent.

Given 1.2 this representation, branch-and-bound proceeds in two steps, as the name suggests. *Branching* refers to navigating our search tree, moving down one level to the next partial assignment. The *bounding* step refers to somehow calculating a bound on the best solution given the partial solution represented by the current node. If the bound is suboptimal relative to an already known solution, we needn't branch on this node; such nodes are said to be *fathomed*. Likewise, we fathom nodes corresponding to infeasible partial assignments.

Once every node has either been branched on or fathomed, we can conclude our search. This process will either identify a maximum independent set or else it will conclusively show that our previously known solution was in fact optimal[11]. Additionally, it is possible to prove upper bounds on our maximum independent sets by assuming a hypothetical solution of a given size. Essentially this technique lies to the algorithm, claiming to have an initial solution of some size. In this case, it is clear that the algorithm will not

conclusively show that our previously known solution was optimal (since it does not necessarily exist), but it does show that any possible solution is no bigger than that value. This strategy corresponds to using branch-and-bound to answer the MIS decision problem for a given value k.

Using branch-and-bound this way points up a general aspect of our discussion so far, which is that the branch-and-bound is a family of algorithms; particular implementations differ in terms of (a) the bounds they use, (b) what order branching occurs (which can be different for different parts of the tree), and/or (c)where our initial solution comes from.

The branch-and-bound algorithm used in the present research uses previously known solutions given in [13], branches in a breadth-first manner, and uses a semi-definite programming relaxation to calculate the upper-bound[1, 2]. We describe the derivation of that bound in the following section.

First, we illustrate the above notation and algorithm by considering the following example.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with $\mathcal{V} = (v_1, ..., v_{12})$ and

$$\begin{array}{rlll}\mathcal{E} = \{ & (v_1, v_2), & (v_1, v_8), & (v_1, v_9), \\ & (v_2, v_9), & (v_2, v_{10}), & (v_2, v_3), \\ & (v_3, v_{10}), & (v_3, v_4), & (v_4, v_{10}), \\ & (v_4, v_{11}), & (v_4, v_5), & (v_5, v_{11}), \\ & (v_5, v_6), & (v_6, v_{11}), & (v_6, v_{12}), \\ & (v_6, v_7), & (v_7, v_{12}), & (v_7, v_8), \\ & (v_8, v_{12}), & (v_8, v_9), & (v_9, v_{12}), \\ & (v_9, v_{10}), & (v_{10}, v_{11}), & (v_{11}, v_{12}) \; \} \end{array}$$

as illustrated in the following figure.

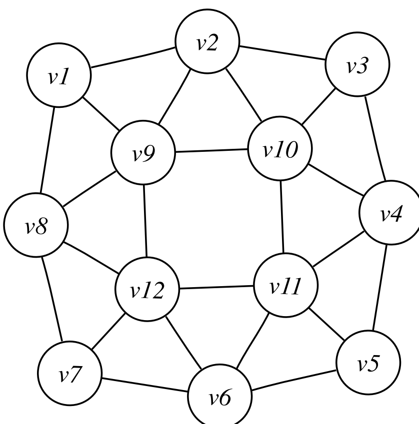


Figure 1.3: Graph for branch-and-bound example.

We note for future reference the obvious symmetry in this example, bilaterally both horizontally and vertically and through rotations of ninety degrees (relative to picture above).

For use in the branch-and-bound algorithm, the set $\mathcal{V}$ is usually indexed by an integer between one and $|\mathcal{V}|$. Which number is assigned to which vertex is arbitrary. In what follows, we use labels corresponding to the diagram above. Similarly, the order in which we enumerate the vertices in our algorithm is also arbitrary, and for this example we do so in ascending order. For example, the partial assignment "010 - - - - - 00 - -" indicates that we have set v_2 into our independent set and excluded all of its neighbors, with the membership status of the remaining vertices undetermined. Such representations of partial assignments are quite convenient and we will refer to them as *assignment strings*. In such situations we say that $\{v_1, v_2, v_3, v_9, v_{10}\}$ are *fixed* and the remaining vertices are *unfixed*. To utilize our previous notation, we say that an assignment string has a 0 in position i if vertex $v_i \in P_0$, and a 1 if

$v_i \in P_1$. We use a "-" to denote $v_i \in U$ (equivalently $v_i \notin P_0 \cup P_1$) just as in figure 1.2.

Our initial solution, determined by inspection, is $\{v_1, v_3, v_5, v_7\}$. Accordingly, when an upper bound is less than or equal to the size of this set (4), we can fathom the corresponding node. For our bound, we rely on the arithmetical observation that given a partial assignment, any independent set "grown" from that assignment can never exceed the number of vertices already in our independent set, plus the number of unassigned vertices. That is, the expression $|P_1| + |\mathcal{V} - (P_0 \cup P_1)|$ gives our bound [2].

We shall additionally augment the branch and bound process by automatically fixing to zero neighbors of vertices fixed at one; this is equivalent to fathoming nodes corresponding to assignments that violate the definition of an independent set (by including adjacent vertices). This procedure is referred to as *constraint propagation* and ensures that we never bother branching on partial assignments that are not independent sets. Such cases correspond to the boxed partial assignments in figure 1.2.

The diagram on the following page illustrates the algorithm as it works its way down the search tree. (We display only the left half of the tree here.) At each step we present the bound and indicate fathomed nodes by grayed boxes.

---

[2] Using this bound we can see that every branching step that sets the branched-upon vertex out of the set will decrease by exactly one. Accordingly, we save a bit of computational effort if we simply don't branch in that direction (to the right in our figure) from nodes with a bound of one greater than our known solution (in this particular case that number is 5).

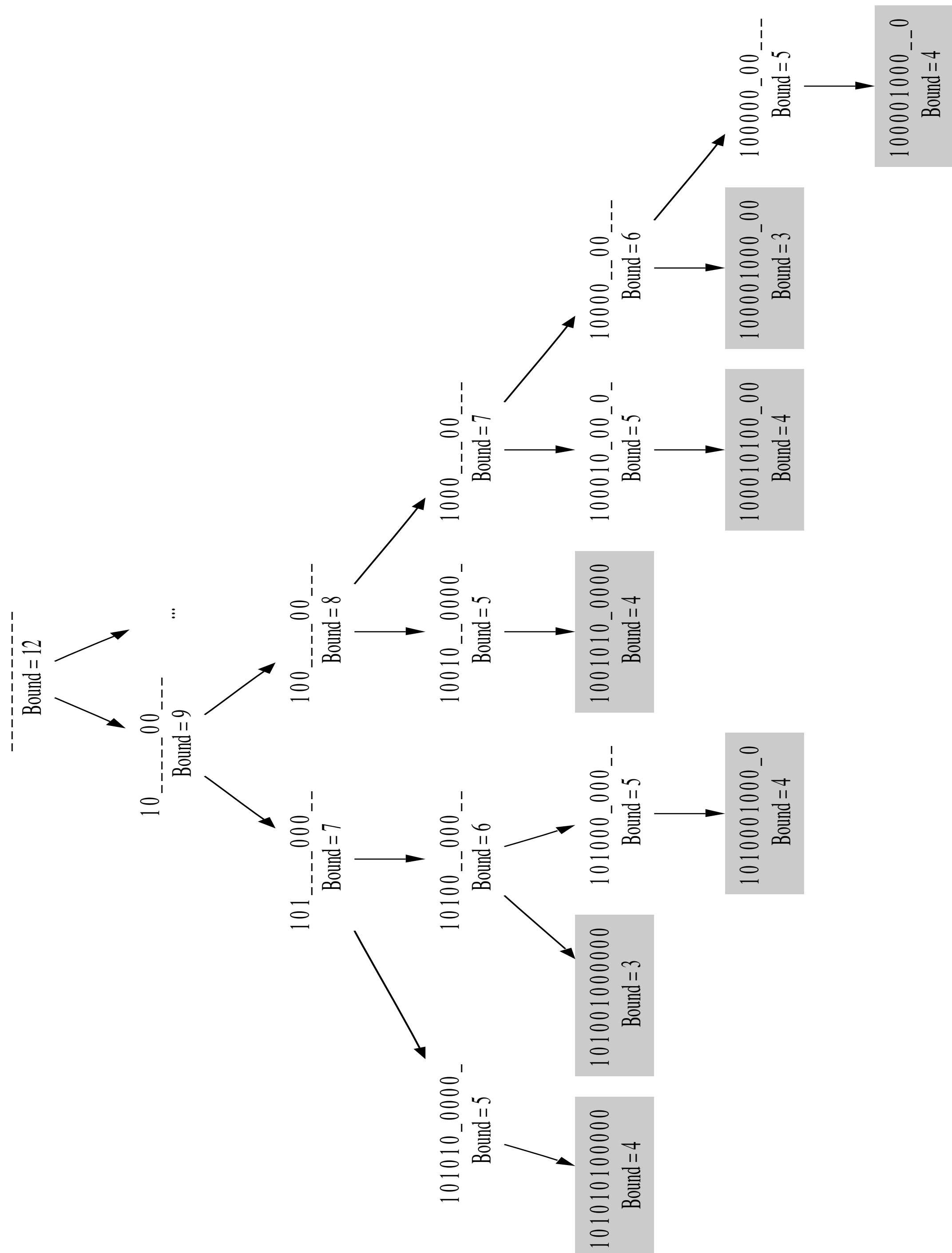


Figure 1.4: Branch-and-Bound Example

1.4 The Lovasz ϑ Bound

As mentioned, at each node in the search tree we must calculate an upper bound on our MIS. The method by which we chose to do so derives from a *relaxation* of the maximum independent set problem. By relaxation we mean that we impose less restrictive constraints than are found in the actual problem, which entails that any optimal solution to the "relaxed" problem is at least as good as the optimal solution to the original problem.

To be specific, let x be the *characteristic vector* of an independent set on graph $\mathcal{G}$. That is, x is a 0-1 vector of length $|\mathcal{V}|$ such that $x_i = 1$ if and only if vertex i is in the independent set.[3] Since x is 0-1, we have that $\sum_i x_i = x^T x$, which equals the size of our independent set. We then consider the matrix $X = \frac{1}{x^T x} x x^T$ which simple computation can show has the following properties.

i. $\mathrm{tr}(X) = 1$

ii. $X \succeq 0$

iii. $X_{i,j} = 0 \;\; \forall (i,j) \in \mathcal{E}$

iv. $\mathrm{tr}(JX) = x^T x$ where J is a matrix of all ones.

Here $X \succeq 0$, read "X is positive semidefinite", means that $y^T X y \geq 0$ for all real-valued vectors y not equal to the zero vector.

[3] In particular, this string is the same as the string described in our discussion of branch and bound enumeration, only here we requiring that P_1 be an independent set and we are treating this string as a vector.

Accordingly, if we drop the restriction that $x \in \{0,1\}^{|\mathcal{V}|}$ and that $X = \frac{1}{x^T x} x x^T$ we arrive at the following semidefinite programing relaxation.

$$\begin{array}{rrcl} \max & \text{tr}\,(JX) & & \text{s.t.} \\ & \text{tr}(X) & = & 1 \\ & X_{i,j} & = & 0 \;\; \forall (i,j) \in \mathcal{E} \\ & X & \succeq & 0 \end{array} \tag{1.1}$$

The upper bound arrived at by this optimization is called the Lovasz ϑ bound, after the author of [8], where the properties of this bound were first explored. There and in subsequent research many different, but ultimately equivalent, formulations have been demonstrated [3, 5, 8, 9]. The particularly intuitive formulation presented above suffices for our purposes and is, in fact, the very one our branch-and-bound code utilizes.

We mention specifically that this bound is substantially stronger than the naive bound featured in our previous example. On that same example, using the Lovasz bound found the MIS in a single stage, at the root node.

1.5 Error-correcting Codes

In the present discussion a *code word* will be a bit string of some fixed length. This restriction is justified since such a two-character alphabet is sufficient to encode anything we might imagine, as modern computing technology vividly demonstrates. Coding theory as an area of mathematical research is motivated by the fact that when transmitting such code words, the bit string that we send is not always the same as the bit string that is received at the other end. The presence of such transmission *errors* naturally raises the question of how we might correctly infer the true – sent – message from the received

– possibly error-riddled – one.

To get a flavor for what this project entails, consider the naive scheme of redundant coding. For every code word we desire to send, we send instead the string that is repeated twice at each bit. For example 0010011 $\rightarrow$ 00001100001111. To see that this facilitates disambiguation, simply notice that any time we receive a string containing a non-duplicated bit, we know immediately that we must have experienced some kind of transmission error.

Of course, if we permit an unlimited number of errors of arbitrary character, then any sent code word could be received on the other end as any other code word. Instead, the theory of error correcting codes [14] considers what kinds of coding schemes are most efficient at guarding against specific sorts of errors. We can quantify efficiency by the number of code words of a certain length necessary to allow errorless interpretation. This measure is appropriate because the size of this set determines how compactly we can express ourselves; literally it is the size of our code vocabulary.

For example, suppose the only error we encounter is that for a given code word of length n we may have a single bit sent incorrectly – a 1 where a 0 ought to have been or vice versa. We can then ask ourselves "which subset(s) of the 2^n possible bit strings of length n can be used so as to prevent the misinterpretation of messages?" For this particular error it can be seen that limiting ourselves to strings with only even numbered adjacent bits (as in the doubling scheme) prevents code word confusion, even if an error strikes. But is this the best we can do? This question of code optimality is one we shall return to shortly. First, we formalize the preceding discussion by introducing

some notation.

We let r index a particular class of error and r_n indicate that class of errors over bit strings of length n. Let s be a bit string and let $T_r(s)$ be the set of bit strings that might be received if s was transmitted. For example, if $s = 01001$ and r is the single-bit-missent error, then $T_r(s) = \{01001, 11001, 00001, 01101, 01011, 01000\}$. (Also, we see that $n = 5$ in this example.)

With this terminology in hand we can now define *error-correcting codes.*

Definition 1.6. *An* **error-correcting code** C_r*, relative to some error class* r*, is a subset of* $\{0,1\}^n$ *such that for every* $s, t \in C_r, T_r(s) \cap T_r(t) = \emptyset$*.*

Now it is easy to see that an optimal code is simply an error-correcting code of maximum size. Such a formulation suggests a connection to maximum independent sets. Indeed, the bridge between the two areas – error-correcting code theory and graph theory – is the *confusion graph* which we now describe.

Definition 1.7. *A* **confusion graph** $\mathcal{G}_r$*, relative to error* r*, is an undirected graph with* 2^n *vertices such that* $(v_i, v_j) \in \mathcal{E}$ *if and only if* $T(s(v_i)) \cap T(s(v_j)) \neq \emptyset$*, where* $s(v_i)$ *is the code word (arbitrarily) associated with vertex* v_i*.*

From this definition, we can, on the one hand, construct the corresponding confusion graph for any particular systematic error we wish to investigate. On the other hand, we can see that any undirected graph on 2^n vertices is a confusion graph *defining* some error. This latter characterization emphasizes

our earlier point about considering arbitrary errors: such a case corresponds to a completely connected graph. Most importantly for our purposes, we have by construction that optimal codes exactly corresponds to maximum independent sets on the confusion graph denoted $\mathcal{G}_{r_n}$.

Our focus at present is a collection of errors identified by Neil Sloane[13]. In particular we are interested in the so-called "1tc" and "1et" errors. These errors can be described as follows.

Definition 1.8. *A* **transposition error** *involves the transposition of any two adjacent bits in a code word. We denote this class of errors by the tag "tc".*

By convention we place a numeral in front of our denoting tags to indicate the number of such errors per code word that we permit; thus "1tc" refers to the class of errors under which one such transposition is considered per code word.

Definition 1.9. *An* **end-around transposition error***, or "et" error is a transposition error where we additionally consider the first and last bits of a codeword to be adjacent.*

On the following pages we present the confusion graphs of 1tc and 1et errors over code words of length four.

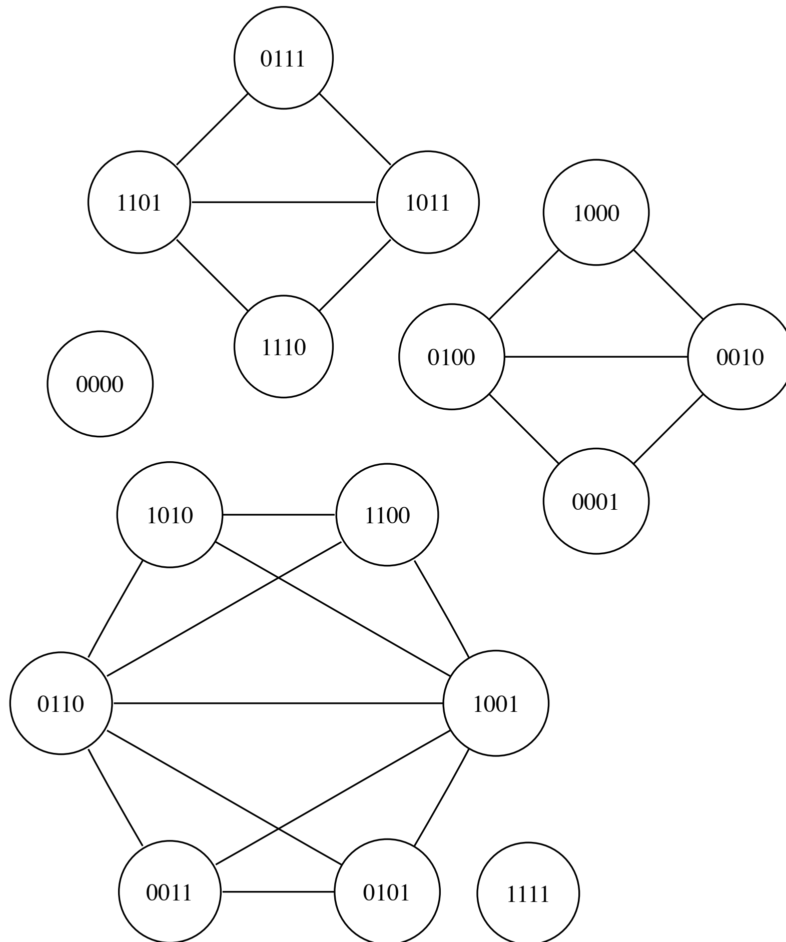


Figure 1.5: The 1tc confusion graph for $n = 4$.

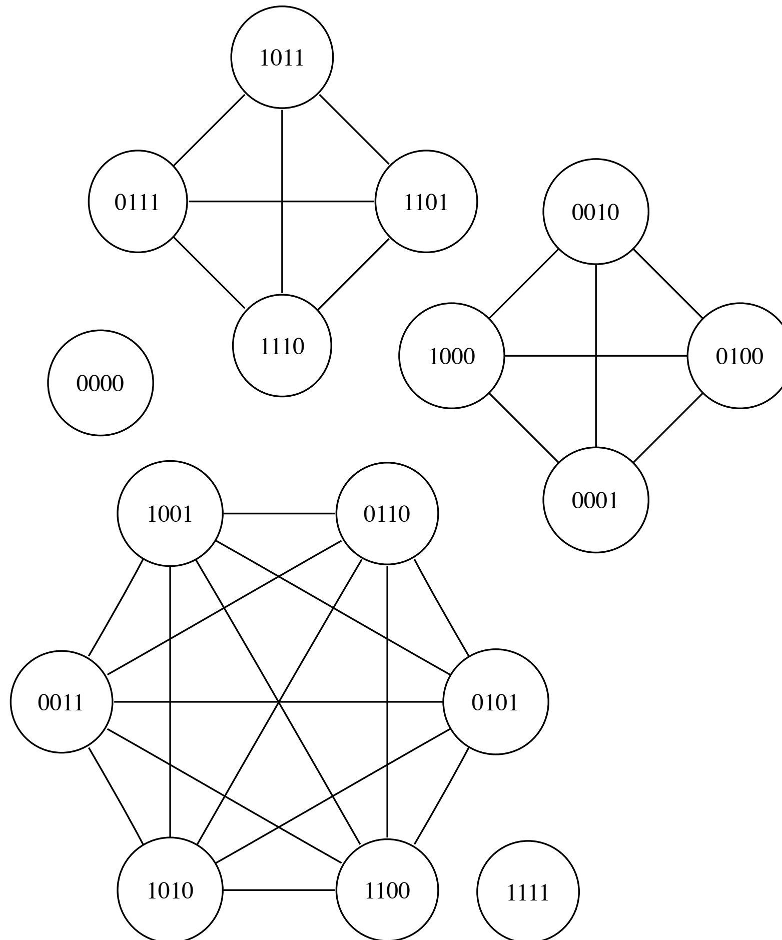


Figure 1.6: The 1et confusion graph for $n = 4$.

We take special note of how these graphs can be decomposed into connected components, subgraphs which are completely disjoint from one another. In the following section we will see how this feature can be used to improve the estimated size of our maximum independent set.

1.6 Augmenting Branch-and-Bound with Additional Information

Informally, one factor contributing to the difficulty of solving NP-complete problems is their great generality. The well-known traveling salesman problem, for example, permits distance matrices of any character whatever. Similarly, the maximum independent set problem is defined over arbitrary graphs. Intuitively, this openness entails that for any given algorithm, perhaps it is possible to dream up an instance of your problem that foils it.

This way of looking at the matter suggests that the more additional information we have about any given problem instance, the better shot we have of solving it efficiently. Indeed, restrictions of NP-complete problems are sometimes markedly easier than their more general cousins. For example, while the satisfiability problem for Boolean formulas in disjunctive normal form is NP-complete, the restricted problem concerning just those formulas having exactly two variables per clause (the 2-SAT problem) can be solved in polynomial time [4].

This idea can be taken further still if we consider applying the principle to all of the sub-problems that one might need to solve in the course of solving a particular larger problem. Of course, we specifically are thinking of branch-and-bound here. In this context, including additional information

can be seen as restrictions on the feasible region over which we are searching. Explicitly adding these constraints at each iteration yields a general approach called branch-and-cut. The "cut" here refers to the region of the solution space that we lopped off by including an additional constraint.

In our branch-and-bound code we make two problem-specific observations that we use to augment our search.

Firstly, we handle connected components of our graphs separately. By this we mean that rather than formulating our Lovasz bound over the entire graph, we do so with respect to the isolated subgraphs. This decision clearly does not affect the outcome of our answer; however, we do benefit in the following way. Consider a graph with two connected components, and suppose that the Lovasz bound for each of them in isolation is 15.7 and 4.4 respectively. Determining this bound jointly we would calculate an upper bound of 20.1 on the size of our maximum independent set. Since a non-integer independent set is absurd, we say that our maximum independent set is no larger than 20. But is this all we can say? By looking at the connected components individually, we notice that a bound of 15.7 for the first component means, by similar reasoning, that our bound is actually 15. Likewise, we can read our 4.4 bound as simply 4. Combining these results after rounding them individually to the next lowest integer gives us a bound improved by one. Since, as we will see later, strengthening our bound by even one vertex can take hundreds of hours of computation, we welcome this improvement.

Secondly, we consider the case where all of a vertex's neighbors are connected to one another – that is, where the neighborhood of a vertex is a

clique. This situation is indicated in the illustration below.

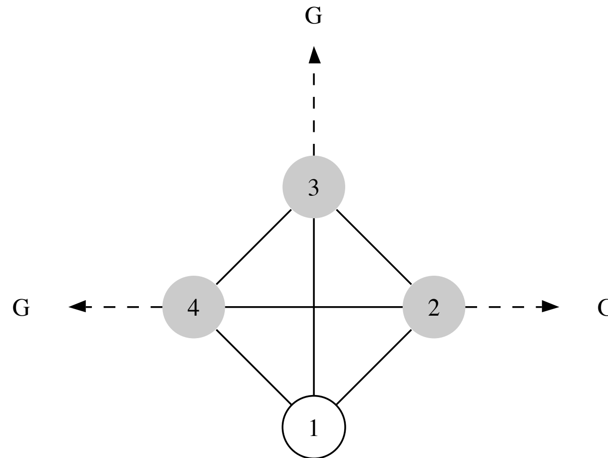


Figure 1.7: Checking for nodes with neighborhoods that are cliques. Node 1 in this example is connected to the clique comprised of vertices 2, 3 and 4 which are themselves then attached the the larger graph. We infer that vertex 1 can safely be included in our maximal independent set.

We note that if a vertex has this property and is *not* in our independent set, then some member of its neighborhood must be. But if one member of its neighborhood is in the set, the rest must not be, in addition to however many vertices are in that vertex's neighborhood. On the other hand, if our original vertex *is* in our independent set, then none of its neighbors can be, but our restrictions end there. That is, we have at least as many vertices in our independent set if we go ahead and automatically add vertices whose neighborhoods are cliques. Accordingly, we check for this property at each step.

In the next section we will see yet another kind of augmenting observation, which constitutes the subject of this thesis.

CHAPTER 2

EXPLOITING SYMMETRY TO SOLVE SLOANE'S MIS CHALLENGE GRAPHS

2.1 Introduction

The key to our algorithm springs from the following definition of symmetry for an undirected graph.

Definition 2.1. *An* **automorphism** *of the graph $\mathcal{G}$ is an edge-preserving permutation π of the vertices $\mathcal{V}$ of $\mathcal{G}$; i.e. $(v_i, v_j) \in \mathcal{E}$ if and only if $(\pi(v_i), \pi(v_j)) \in \mathcal{E}$.*

That is, π is a one-to-one and onto mapping $\pi : \mathcal{V} \rightarrow \mathcal{V}$ that leaves the graph unchanged except for the labels, or names, of the vertices. In terms of our earlier example we see that

$$\begin{aligned}
\pi : \mathcal{V} \rightarrow \mathcal{V} &= \{ \\
v_1 &\rightarrow v_7 \\
v_2 &\rightarrow v_6 \\
v_3 &\rightarrow v_5 \\
v_4 &\rightarrow v_4 \\
v_5 &\rightarrow v_3 \\
v_6 &\rightarrow v_2 \\
v_7 &\rightarrow v_1 \\
v_8 &\rightarrow v_8 \\
v_9 &\rightarrow v_{12} \\
v_{10} &\rightarrow v_{11} \\
v_{11} &\rightarrow v_{10} \\
v_{12} &\rightarrow v_9 \\
& \}
\end{aligned}$$

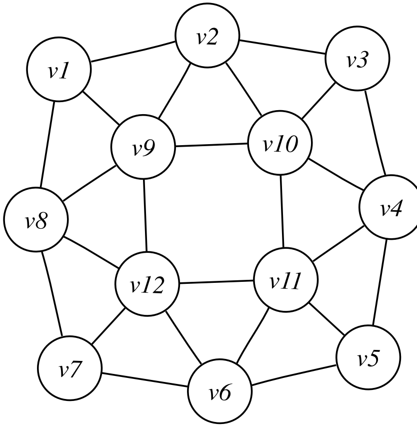


Figure 2.1: Branch-and-bound example graph revisted.

is an edge preserving permutation (one of many in this example). Before continuing, the following definition will prove useful.

Definition 2.2. *Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and an automorphism π on $\mathcal{G}$, an* **orbit** *is a sequence of vertices that map into each other one after another under iterations of π.*

In the general group theoretic definition of an orbit, π can be any group acting on an arbitrary set S. The orbit of a given element $s \in S$ is then the set of all elements to which s can be moved by elements of π.

With a particular automorphism π identified, we can make the following simple observation: given an orbit under our automorphism, we needn't consider assignments on the vertices in this orbit that are indistinguishable from assignments that have already been considered. That is, for any independent set I, we also have an independent set $\pi(I)$. Since I and $\pi(I)$ clearly have

the same size, if our only concern is the size of the maximal independent set, we need not consider both of them.

This observation becomes particularly useful in the context of branch-and-bound, if we let our automorphism π act on partial assignments. A small example should help fix this idea.

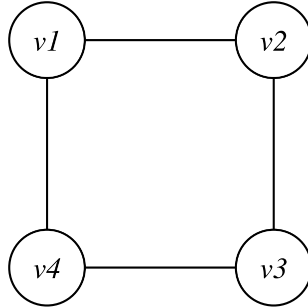


Figure 2.2: Example graph revisited to illustrate orbits and symmetry.

Consider the four-vertex graph shown above: $\mathcal{V} = \{v_1, v_2, v_3, v_4\}$, $\mathcal{E} = \{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_3, v_4)\}$. We note initially two orbits, each of size two, corresponding to the automorphism

$$\begin{array}{rcl} \pi : \mathcal{V} \to \mathcal{V} & = & \{ \\ v_1 & \to & v_2 \\ v_2 & \to & v_1 \\ v_3 & \to & v_4 \\ v_4 & \to & v_3 \\ & & \}. \end{array}$$

Consider now the orbit $(v1, v2)$. There are four possible assignments on this orbit, corresponding to partial assignments on the graph of 00 - -, 11 - -, 10 - - and 01 - -. Thus we see that $\pi(10 - -) = 01 - -$ and infer that in the

course of our branch-and-bound process, it is unneccesary to check both of these assignments.

As it turns out, we can extend this idea in two directions. Firstly we would simply like to consider orbits of the largest size possible. Secondly, we would like to iterate the process as much as possible. By this we mean that if a partial assignment is unchanged by our symmetry, we want to go on to consider yet another orbit. For example, in the little graph above the assignments 00 - - and 11 - - are both preserved under our automorphism π. That is:

Definition 2.3. *A partial assignment is* **compatible** *with an automorphism* π *if we have that for all* $v_i \in \mathcal{V}$, $v_i \in I$ *if and only if* $\pi(v_i) \in I$ *and also* $v_i \in U$ *if and only if* $\pi(v_i) \in U$.

Notice that since our enumeration for branch-and-bound (that is, the ordering of our assignment string) is arbitrary, we can preferentially search through partial assignments so as to deliberately encounter compatible partial assignments.

Using these techniques it is possible to reduce the size of our search tree during branch-and-bound. Presently we demonstrate by applying these ideas to our earlier example. Notice in particular that we grow our tree in increments equal to the size of our orbit. Let us call this number b, which in our example is equal to 2. We note at this time that growing or navigating our search tree in "chunks" like this is equivalent to searching only unique partial assignments at level b in our full tree.

A quick comparison of the search trees should illustrate the improvement; for parity we show only one half of the search tree as before.

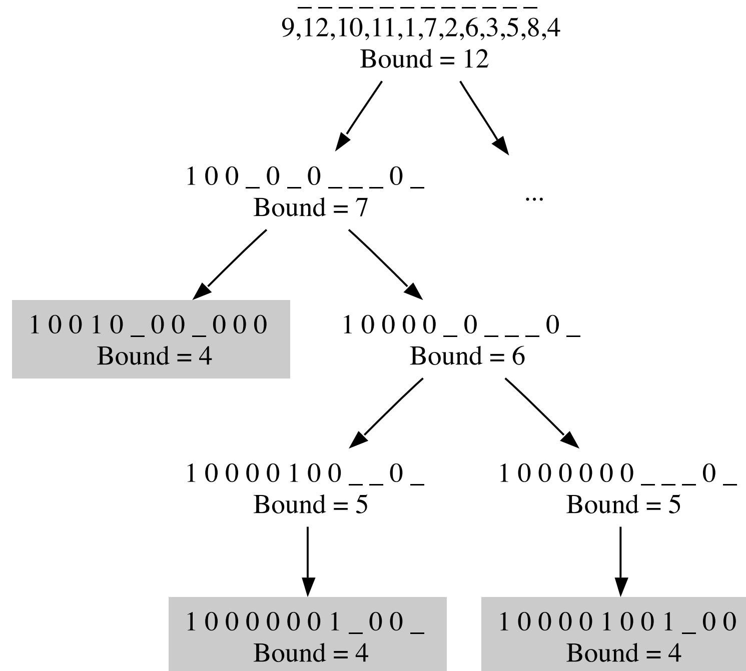


Figure 2.3: Branch-and-Bound Example With Symmetry Breaking and Constraint Propagation

In the next chapter we consider how much computational savings can be attained by employing these techniques and present our actual computational results.

2.2 Symmetry in 1et and 1tc Error Graphs

As it turns out, Sloane's graphs have many symmetries. In particular, because their construction is given by a simple set-theoretic rule, we can analytically derive automorphisms with respect to the underlying vertex labels, the n-bit code words. In general we have the following theorem.

Theorem 2.1. *For a confusion graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, if some permutation π commutes with T, then π is an automorphism on $\mathcal{G}$.*[1]

Proof. Recall that $T(s)$ returns the possible set of received code words when code word s is sent (relative to error r) and that $s(v_i)$ is the code word associated with vertex v_i in our confusion graph. The theorem then claims that for coding confusion graphs, if $\pi(T(s(v_i)) = T(\pi(s(v_i)))$ for all v_i (if π and T commute), then π is a automorphism on our confusion graph.

For coding graphs we have that $(v_i, v_j) \in \mathcal{E}$ if and only if $T(s(v_i)) \cap T(s(v_j)) \neq \emptyset$. So if $\pi(T(s(v_i))) = T(\pi(s(v_i)))$ for all i we have

$$\pi(T(s(v_i)) \cap T(s(v_j)))) = \pi(T(s(v_i))) \cap \pi(T(s(v_j))) = T(\pi(s(v_i))) \cap T(\pi(s(v_j))).$$

Since $\pi(A) = \emptyset$ if and only if $A = \emptyset$ for all sets A, our result follows. □

Using this result, we are able to demonstrate particular automorphism groups over our coding graphs. First we look at the 1et error described earlier. We first describe the bit-shift permutation.

Definition 2.4. *The* **rotation** *permutation, denoted here as ρ, is a mapping from strings $s = s_1 s_2 s_3 \ldots s_n \rightarrow s' = s_n s_1 s_2 \ldots s_{n-1}$. If one considers a string as a value-index pair, for every tuple, ρ takes index $i \rightarrow i + 1 \pmod{n}$.*

Theorem 2.2. *The 1et error commutes with the rotation permutation.*

[1] We have dropped r and n from our subscripts for sake of neatness; it is understood that the error class r is predefined and entirely described by the graph $\mathcal{G}$ and that n is a fixed integer.

Proof. Define $t_{1et}^{(k)}(s)$ to be a permutation of the indices of the string s such that

$$\begin{array}{lllll} i & \to & i+1 & \pmod{n} & \text{if } i = k \\ i & \to & i-1 & \pmod{n} & \text{if } i = k+1 \\ i & \to & i & & \text{otherwise} \end{array}$$

Then we can write $T_{1et}(s) = \{t_{1et}^{(k)}(s) \text{ for } k \in \{1, 2, \ldots n\}\}$. Since $\rho(s)$ permutes the indices of s such that $i \to i+1 \pmod{n}$, a simple substitution shows that

$$t_{1et}^{(k)}(\rho(s)) = \rho(t_{1et}^{(k-1 \pmod{n})}(s)).$$

Since $T_{1et}(s)$ contains $t_{1et}^{(k)}(s)$ for all $k \in \{1, 2, \ldots n\}$ we see that

$$T_{1et}(\rho(s)) = \rho(T_{1et}(s))).$$

□

This automorphism has an orbit of size n and because of its relatively large size, which grows with the length of the code words, the symmetry finding method is quite helpful for 1et graphs.

The ρ permutation is *not* an automorphism with respect to 1tc graphs, because by not permitting the "wrap around" transposition, we must get rid of the modulus in the corresponding definition and as a result, the above proof no longer works. In fact, finding a counterexample is a simple matter. For the string 001 we find that $T_{1tc}(\rho(001)) = \{010, 100\}$ while $\rho(T_{1tc}(001)) = \{100, 001\}$.

Instead, for the 1tc error and corresponding confusion graphs, we use a smaller-orbit automorphism.

Definition 2.5. *The* **string flip** *permutation, denoted here as* σ *, flips the string around by permuting the indices (i) of the a string s such that* $i \rightarrow n - i + 1$.

It is easy to see that since the string flip permutation preserves adjacency, it commutes with the single transposition error. Similarly, the orbit is just of size two, since flipping a string around twice gets you back to the original string, i.e. $n - (n - i + 1) + 1 = i$.

2.3 Computational Results

Being a relatively flexible framework, it can be a tricky matter to characterize improvements to a branch-and-bound algorithm. For example, one must weigh the advantage of a sharper bound against the computational effort to achieve that bound. On the one hand, a tighter bound might lead to more fathoming and so offset the per-node evaluation time it might add. On the other hand, a tighter bound, despite being an improvement, might fail to impact the frequency of fathoming. Additionally, certain modifications to a branch and bound algorithm might be quite sensitive to order of branching or another essentially free parameters. Determining how these choices interact with a particular graph is not necessarily, therefore, a systematic matter. That said, we can venture some statements, and, naturally, empirical improvement cannot be ignored: our method has solved problems that had previously gone unsolved.

The most straightforward way to quantify the improvement in our algorithm is to consider how many assignments we *do not* have to consider

under our symmetry arguments that otherwise might have occupied the algorithm. This analysis is imperfect for the sorts of reasons discussed in the previous paragraph. In particular, this measure of improvement is making the assumption that *all* our nodes would eventually have been visited. This is a patently untrue assumption, as it completely ignores the bounding aspect of the process. More insidiously, this approach requires beginning the tree at level b, the length of our orbit. Should we, in fact, have realized significant pruning of our tree at levels prior to this, our method can actually be out-performed by unmodified branch-and-bound. With these shortcomings in mind, however, we can provide an upper bound on the improvement we hope to see.

The fundamental trick to our approach, and also to analyzing it, is to take advantage of the fact that the order of the nodes in our binary tree is arbitrary. By ordering them relative to the orbits under an automorphism π we are able to regulate the dynamics of our partial assignments under iterations of π. In particular, listing them by orbit, one after another, in order, guarantees that the partial assignment string is shifted by one bit to the right (with wrap around) at each iteration of π. Thus ordered, we can ask the following question: how many of these partial assignments represent unique underlying assignments?

It turns out that this question has already been studied in some detail [15]. Typically the question is phrased like this: what is the number of distinct necklaces that can be made out of a fixed number of just two kinds of beads, disallowing flipping the necklace over? The equivalence should be clear: the number of beads is equal to b, the size of our orbit, and our beads are ones and

zeros. The formula for this number is given by

$$Z(b) = \frac{1}{b}\sum_{d|b}\phi(d)2^{b/d} \tag{2.1}$$

where $\phi(b)$ is Euler's totient function defined as the number of positive integers less than n that are relatively prime to n.

So, beginning at level b, searching through all assignments would require that we search each of the 2^b root nodes and the associated binary trees below them. Taking into account the underlying symmetry we find that we need only consider $Z(b)$ out of the 2^b total subtrees. The ratio between these two numbers is the factor of our savings in terms of orbit size b. The plot

Table 2.1: Theoretical Savings Estimate

b	$Z(b)$	$2^b/Z(b)$
2	3	1.33
3	4	2.00
4	6	2.66
5	8	4.00
6	14	4.57
7	20	6.40
8	36	7.11
9	60	8.53
10	108	9.48
11	188	10.89

below shows that this relationship is strongly linear; in fact it shows that the anticipated savings is approximately a factor of b. As it happens, this theoretical estimate matches quite closely the actual improvements we witnessed on Sloane's 1et.2048 and 1tc.2048 graphs. The relevant details are depicted in the following charts.

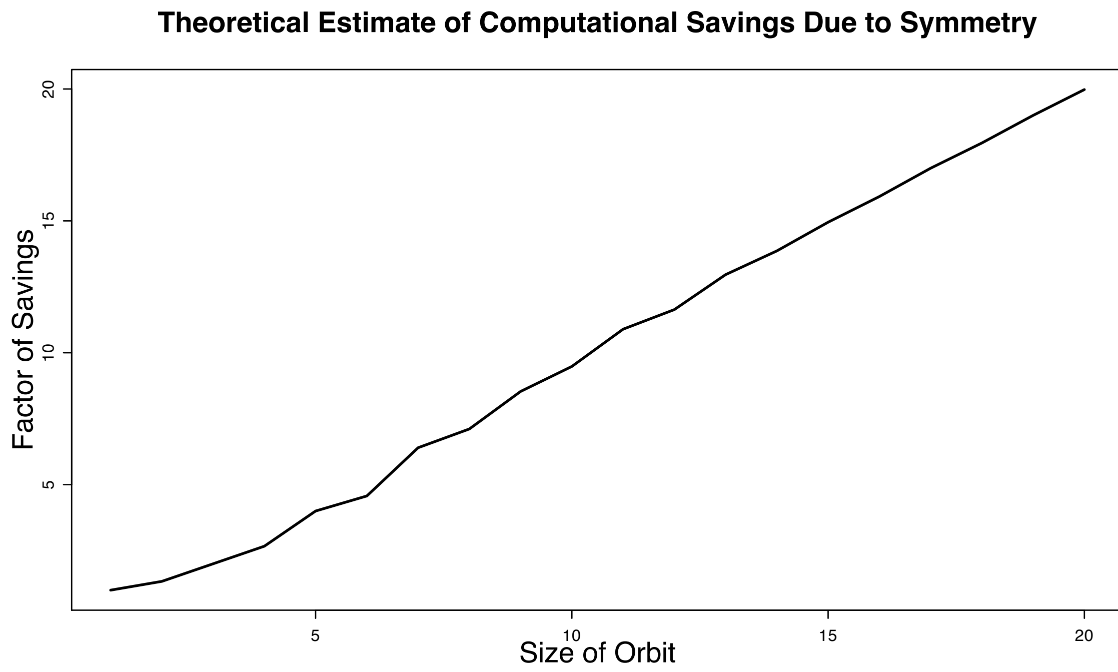
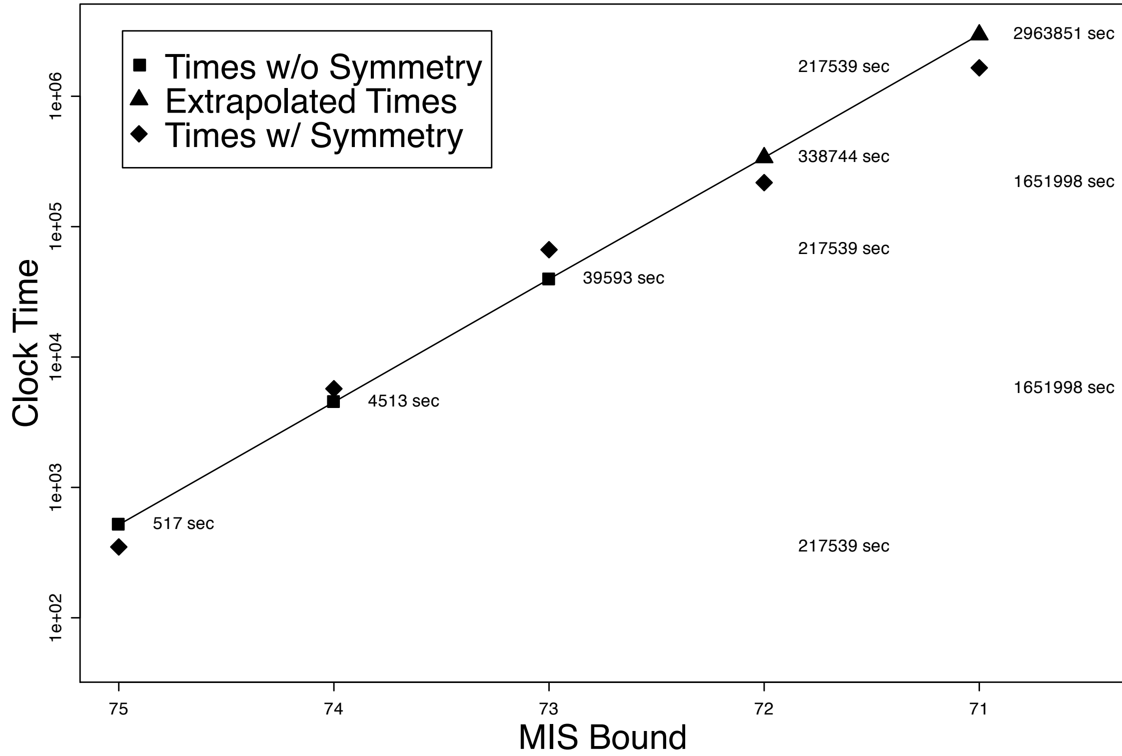


Figure 2.4: Factor of Savings From Identifying Symmetry



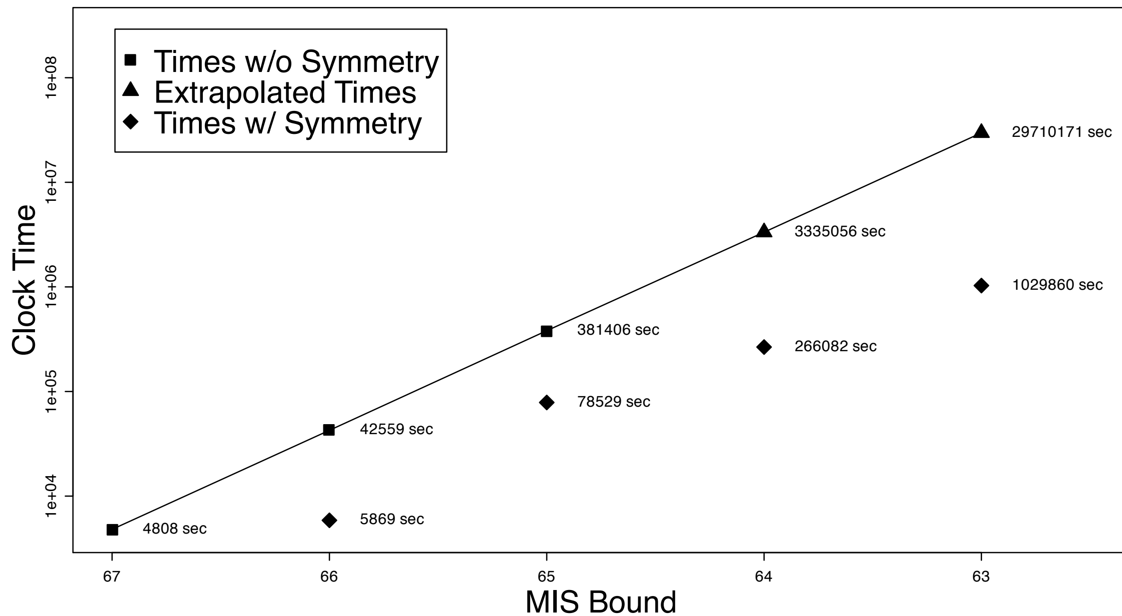


Figure 2.5: Computational Results for 1tc.2048 and 1et.2048

Table 2.2: Computational Savings for 1et.2048

MIS Bound	Time w/o Sym.	Time. w/ Sym.	Factor of Savings
66	42559	5869	7.25
65	381406	78539	4.86
64	3335056	266083	12.53
(optimal) 63	29710171	1029860	28.84

These plots make apparent that the larger orbit found in the 1et graphs allows for greater savings, consistent with the theoretical prediction that the savings will be proportional to orbit length. In fact, our savings exceeded this estimate in both cases. We realized a savings factor of approximately 1.8 versus the theoretical 1.33 on the 1tc graph, and approximately 29 versus the theoretical 11 on the 1et graph. (We note also that in the 1tc graph for the looser approximations the symmetry scheme actually takes more time; this reflects the overhead involved with using separate starting files, etc.)

Some of this improvement over the theoretical estimate comes from the fact that of the 188 unique assignments on our size 11 orbit, only 6 of them correspond to independent sets.

CHAPTER 3

APPLICATIONS TO ARBITRARY GRAPHS

3.1 Finding Symmetries

We close now with a look at how our results might be applied to arbitrary graphs. As we have mention above, one aspect of what makes NP-complete problems so difficult is the extreme generality of their statement. Accordingly, we might expect that making the symmetry techniques used here more general would likewise be difficult. Indeed, it turns out that finding symmetries in arbitrary graphs is itself a difficult problem.

In general, this problem is called the *graph isomorphism problem.* Its decision problem reads:

> Given graphs $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$ and $\mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2)$ does there exist a mapping $f : \mathcal{V}_1 \to \mathcal{V}_2$ such that $(x, y) \in \mathcal{E}_1$ iff $(f(x), f(y)) \in \mathcal{E}_2$?

Clearly finding an automorphism is just the case where $G_1 = G_2$, (which is only interesting if we exclude the identity permutation).

The graph isomorphisms problem is obviously in NP, since given a particular mapping, checking if it satisfies the criteria is a straightforward matter. Furthermore, while the problem is hard, it is not known whether or not it is NP-complete. In fact, this problem is used to define its own complexity class of

GI-complete problems, a class that is thought to be disjoint from NP-complete and P [12].

Fortunately, it seems that in practice the problem is tractable, and it is only on a relatively pathological subset of graphs that graph isomorphism stymies known methods. This fact thus suggests the following straightforward plan for tackling arbitrary graphs with our symmetry-expoiting branch-and-bound:

1. Check $\mathcal{G}$ for automorphisms.
2. Design a branching scheme using orbits from one of these automorphism (as described earlier).

To check the practicability of this method we tested it on Sloane's 1et.2048 graph as well as a few benchmark graphs. It is interesting to look first at the results for our 1et graph, for which we have an analytically determined symmetry that we have already seen yields a $\times 28$ speed-up. Here we list a maximally sized orbit founding using Brendan McKay's excellent NAUTY (No AUtomorphism, Yes?) algorithm [10] as well as one under the rotation automorphism.

Using this size 22 orbit we know that $Z(22) = 190,746$. As with the 11 orbit case, many of the unique assignments are infeasible; in this case, of the 190,746 unique assignments we find that only 3,527 of them correspond to independent sets. However, using these start files in our branch-and-bound code was only marginally faster than running the code alone when we tested it to a bound of 66: using our size-22 orbit took 10 hours, 36 minutes and 26

Table 3.1: Orbits in 1et.2048

NAUTY Orbit	Rotation Orbit
00000110111	00000110111
00000111011	10000011011
00001101110	11000001101
00001110110	11100000110
00011011100	01110000011
00011101100	10111000001
00110111000	11011100000
00111011000	01101110000
01100000111	00110111000
01101110000	00011011100
01110000011	00001101110
01110110000	00000110111
10000011011	
10000011101	
10110000011	
10111000001	
11000001101	
11000001110	
11011000001	
11011100000	
11100000110	
11101100000	

seconds to achieve a bound of 66, whereas straight branch-and-bound took 11 hours, 49 minutes and 19 seconds. This is only 1.125 times faster compared to over 7 times faster with the smaller orbit. This vividly demonstrates the trade-off involved with beginning branch-and-bound at a lower level of the tree; we can see that between level 11 and 22 our code was able to fathom significant numbers of nodes. Moreover, in these comparisons we have not included the not insubstantial amount of time it requires to isolate the appropriate starting assignments.

The final question that remains is whether or not we can expect to find helpful symmetries in arbitrary graphs. To investigate this, we considered the complement graphs of DIMACS benchmark graphs for the maximum clique problem. Notice that maximum cliques corresponds exactly to maximum independent sets in the complement graphs. We consider only graphs with less than 5,000 edges, due to memory constraints of our semidefinite programming solver. Utilizing newer limited-memory methods for semidefinite optimization will expand the applicability of the methods so-far discussed. Using NAUTY, we produced the following table.

Table 3.2: DIMACS Benchmark Graphs

Graph Name	|Max Orbit|	#
david	1	-
DSJC125.5	1	-
DSJC125.9	1	-
DSJC250.9	1	-
DSJR500.1c	2	1
huck	8	1
jean	7	1
miles1000	3	2
miles1500	6	2
myciel3	5	2
myciel4	5	4
myciel5	5	8
myciel6	5	16
queen5_5	8	1
queen6_6	8	3
queen7_7	8	3
queen8_12	4	24
queen8_8	8	6
queen9_9	8	7

Clearly many of these graphs have symmetries. Whether or not these symmetries confer any computational benefit is another matter. Indeed, we find that our branch-and-bound without symmetry deals with these graphs handily, as shown below. On the single graph that took any substantial amount of time – DSJC250.9 – there turns out to be no symmetry to exploit.

Table 3.3: DIMACS Benchmark Graphs

Graph Name	Nodes Processed	Seconds
david	1	5
DSJC125.5	29	23
DSJC125.9	271	18
DSJC250.9	626061	1860
DSJR500.1c	33	70
huck	111	46
jean	117	17
miles1000	1	17
miles1500	1	5
myciel3	1	0
myciel4	1	0
myciel5	1	0
myciel6	1	7
queen5_5	1	3
queen6_6	1	0
queen7_7	1	0
queen8_12	1	4
queen8_8	1	0
queen9_9	1	1

These results and our earlier results from the Sloane's graphs demonstrate that the savings available from considering symmetry range from substantial to non-existent.

CHAPTER 4

Conclusions

A first remark is that the asymptotic complexity of branch-and-bound for the MIS problem on these graphs is still exponential. Even though we were able to realize a 28-fold improvement in some cases, this constant factor savings will be dwarfed as we move to consider ever larger graphs.

With that caveat in mind, however, there are three basic conclusions one can draw from this work.

Firstly, symmetry can be useful in decreasing the necessary search space when solving combinatorial optimization problems. Our main achievement has been to prove the size of the maximum independent set in Sloane's 1et.2048 and 1tc.2048 graphs from coding theory.

Our second conclusion is that the applicability of this technique is limited to cases where symmetry obtains *and* the graph is otherwise difficult. Comparing Sloane's graphs to our DIMACS sample graphs suggests that this will more likely be the case when the graphs are produced according to some underlying rule. Our DIMACS experimentation demonstrates that one cannot expect to find useful symmetry in generic graphs.

Finally, we showed that the NAUTY heuristic of Brendan McKay successfully identifies symmetry in arbitrary graphs and that this information can easily be parlayed into a branching scheme that prevents redundant search.

This is an important point because it means that if indeed we have graphs that are likely to have symmetry, we need not rely on our own cleverness to locate these patterns.

Bibliography

[1] Brian Borchers. CSDP, a C library for semidefinite programming. *Optimization Methods and Software*, 11(1):613–623, 1999.

[2] Brian Borchers. Solving Sloane's independent set challenge problems. Seminar Talk, 2006.

[3] Igor Dukanovic and Franz Rendl. Semidefinite programming relaxations for graph coloring and maximal clique problems. *Mathematical Programming*, 109(2-3):345–365, March 2007.

[4] Michael R. Garey and David S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[5] Gerald Gruber and Franz Rendl. Computational experience with stable set relaxations. *SIAM Journal of Optimization*, 13(4):1014–1028, 2003.

[6] David S. Johnson and Michael A. Trick, editors. *Cliques, Coloring, and Satisfiability*. American Mathematical Society, 1996.

[7] Donald L. Kreher and Douglas R. Stinson. *Cominatorial Algorithms: Generation, Enumeration, and Search*. CRC Press, 1999.

[8] L Lovasz. On the Shannon capacity of a graph. *IEEE Trans. Inform. Theory*, 25:1–7, 1979.

[9] L Lovasz and A. Schriver. Cones of matrices and set-functions and 0-1 optimization. *SIAM Journal of Optimization*, 1(2):166–190, May 1991.

[10] Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[11] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization*. Dover, 1998.

[12] S. Skiena. *Implementing Discrete Mathematics: Combinatoric Graph Theory with Mathematica*, chapter "Graph Isomorphism", 5.2, pages 181–187. Addison-Wesley, 1990.

[13] N.J.A. Sloane. Challenge problems: Independent sets in graphs. http://www.research.att.com/ njas/doc/graphs.html.

[14] N.J.A. Sloane. *The Theory of Error-Correcting Codes*. North-Holland, 3rd edition, 1981.

[15] N.J.A. Sloane. On single-deletion-correcting codes. Technical report, Information Sciences Research, AT&T Shannon Labs, 2002.