Hamiltonian Cycles in Symmetric Graphs

Dov Zazkis

B.Sc., Simon Fraser University, 2006

Thesis Submitted In Partial Fulfillment Of

The Requirements For The Degree Of

Master of Science

in

Mathematical, Computer, and Physical Sciences

(Mathematics)

University of Northern British Columbia

November 2008

Your file Votre référence
ISBN: 978-0-494-48775-4
Our file Notre référence
ISBN: 978-0-494-48775-4

Abstract

Let k be a positive integer. We define M_k to be the graph with a vertex set consisting of all binary strings of length $2k + 1$ which have either k or $k + 1$ ones and edge set consisting of all pairs of these binary strings which differ in exactly one bit. Showing that the graph M_k is Hamiltonian for all k is known as the Middle Levels problem. This problem was first posed in the early 1980's and to this day remains unsolved. In this thesis we explore the symmetries of M_k and graphs related to it. We then use these symmetries to propose a method for finding Hamiltonian cycles in M_k when $2k + 1$ and k are prime. We believe that our method is more efficient than methods proposed by previous authors..

Acknowledgements

I'd like to thank my supervisor, Dr. Iliya Bluskov, for his guidance, support and for introducing me to the Middle Levels problem which I have enjoyed working on so much. I'd like to thank Dr. Jennifer Hyndman, for helping me negotiate through some of the group theory problems which unexpectedly popped up during the writing of this thesis. I'd like to thank my mother, Dr. Rina Zazkis, for her endless emotional and almost endless financial support. Finally, I'd like to thank Heinrich Butow, Eric Chlebek and Ashton Fedler for helping me with coding of my algorithm.

Contents

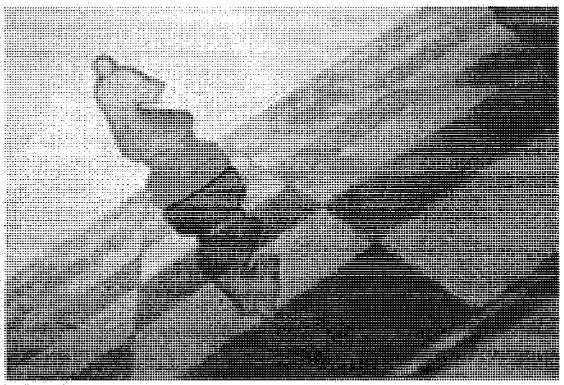
List of Figures

List of Tables

Chapter 1

Introduction

A **graph** G is defined as a pair of sets $(V(G), E(G))$, where $V(G)$ is a set of objects called **vertices** and $E(G)$ is a set of unordered pairs of vertices from $V(G)$ called **edges**. A **Hamiltonian cycle** in G is a list of vertices $v_1, v_2, v_3, ...v_n$ such that every vertex in $V(G)$ appears in the list exactly once and $v_iv_{i+1}, v_1v_n \in E(G)$ for $1 \leq i < n$. Studying the existence of Hamiltonian cycles in graphs has been an important part of graph theory since its inception. Although graphs were not explicitly mentioned until 1878, some examples of problems equivalent to finding Hamiltonian cycles in certain graphs date back hundreds of years. The knights tour problem is such an example. It involves finding a sequence of moves on a chessboard which allows a knight to visit each of the 64 squares on that chessboard exactly once and return to the starting point.



This problem can be represented as a graph: Let each vertex represent a square on the chessboard and for each pair of vertices, v_1, v_2, let v_1v_2 be an edge in the graph if and only if the squares which v_1 and v_2 represent can be reached from one another by a knight in one move. A Hamiltonian cycle in this graph is a solution to the knights tour problem. Systematic solutions to this problem were published by Leonhard Euler in 1759 [10].

The studies of Hamiltonian cycles in graphs can be roughly split into two main categories of problems: 1) Finding sets of conditions which force the existence of sufficiently many edges which in turn forces the existence of a Hamiltonian cycle, 2) Using symmetries as well as structural characteristics to show that certain classes of graphs have Hamiltonian cycles. This is typically done recursively, using Hamiltonian cycles in smaller graphs to build Hamiltonian cycles in larger graphs. For convenience, we will refer to the first type as "density based" and the second type as "non-density based" results. Our main focus will be the latter.

Density based theorems are limited in that they can not be used to show that certain classes of graphs which have a relatively low number of edges have a Hamiltonian cycle. For example, none of the density based theorems can be used to show that C_k with $k > 6$ (the cycle on k vertices) is Hamiltonian, even though it is trivial to show that C_k is Hamiltonian. There are many classes of graphs which are believed to be Hamiltonian for which density based theorems can not be used to show the existence of a Hamiltonian cycle. These graphs are typically quite symmetric. To show that graphs which are believed to be Hamiltonian are in fact Hamiltonian, it becomes necessary to use the properties of these graphs to either explicitly construct a Hamiltonian cycle or show its existence using other non-density based methods. These non-density based methods are also limited in that all current known results are difficult to apply to a wide variety of graphs. There are many open problems which involve proving that certain classes of highly symmetric graphs are Hamiltonian.

Chapter 2

Density Based Results

Even though it is not our focus, it is reasonable to start with some density based results to give the reader a sense of what kind of results can be applied generally to all graphs to test whether they are Hamiltonian, regardless of symmetries. We will also include some well known group theory results in this section which will be needed in later proofs.

2.1 Definitions

Definition 2.1.1 *A **graph**, G, is a pair $(V(G), E(G))$, where $V(G)$ is a set of points called **vertices** and $E(G)$ is set of unordered pairs of vertices from $V(G)$.*

Definition 2.1.2 *Two vertices, v_1 and v_2 in $V(G)$ are said to be **adjacent** if $v_1v_2 \in E(G)$.*

Definition 2.1.3 *A **subgraph**, $G' = (V'(G), E'(G))$, of a graph $G = (V(G), E(G))$ is a graph such that $V'(G) \subseteq V(G)$ and $E'(G) \subseteq E(G)$ with $xy \in E'(G)$ only if $x, y \in V'(G)$. The subgraph is said to be a **spanning** subgraph if $V'(G) = V(G)$.*

Definition 2.1.4 *The subgraph of $G = (V(G), E(G))$ **induced** by the set $W \subseteq V(G)$ is the subgraph $G' = (V'(G), E'(G))$ with $V'(G) = W$ and $E'(G) = \{v_iv_j | v_i, v_j \in V'(G)$ and $v_iv_j \in E(G)\}$.*

Definition 2.1.5 *A graph $G = (V(G), E(G))$ is said to be **bipartitie** if $V(G) = X \cup Y$ with $|X \cap Y| = 0$, and if $v_1v_2 \in E(G)$, then exactly one of v_1, v_2 is in X.*

Definition 2.1.6 *A graph G is said to be **Hamiltonian** if it contains a Hamiltonian cycle.*

Definition 2.1.7 *A **Hamiltonian path** in a graph $G = (V(G), E(G))$ is a list of vertices $v_1, v_2, v_3, ...v_n$ such that every vertex in $V(G)$ appears exactly once in the list and $v_iv_{i+1} \in E(G)$ for $1 \leq i < n$.*

Definition 2.1.8 *A **path** in a graph $G = (V(G), E(G))$ is a list $v_1, v_2, v_3, ..., v_k$ such that $v_iv_{i+1} \in E(G)$ for $1 \leq i < k$ and for all $1 \leq i, j < k$ $v_i \neq v_j$. The vertices v_1 and v_k are referred to as the* **endpoints** *of the path.*

Definition 2.1.9 *A graph G is said to be $k-$**connected** if for every pair of vertices v_1, v_2 the graph contains at least k disjoint paths with v_1, v_2 as endpoints. In general, we refer to $1-$connected graphs as* **connected** *graphs.*

Definition 2.1.10 *We say a set $X \subseteq V(G)$ is **independent** if there are no edges between vertices in X. The cardinality of the largest independent set in G is called* *the* ***independence number*** *and is denoted* $\alpha(G)$.

Definition 2.1.11 *The* ***degree*** *of a vertex* v*, denoted* $\delta(v)$ *is the number of edges incident with it. The minimum and maximum degree of a graph* G *are defined as* $\delta_{min}(G) = min_{v \in V(G)}\delta(v)$ *and* $\delta_{max}(G) = max_{v \in V(G)}\delta(v)$*, respectively.*

Definition 2.1.12 *A graph* G *is said to be* m*-**regular** if* $\delta(v) = m$ *for all* $v \in V(G)$.

Definition 2.1.13 *Two graphs,* $G = (V(G), E(G))$ *and* $H = (V(H), E(H))$ *are said to be* ***isomorphic*** *if there exists a bijective mapping* $\gamma : V(G) \mapsto V(H)$ *such that if* $v_1, v_2 \in V(G)$ *and* $\gamma(v_1) = v_1'$, $\gamma(v_2) = v_2'$ *then* v_1 *and* v_2 *are adjacent if and only if* v_1' *and* v_2' *are adjacent.*

Definition 2.1.14 *Two graphs,* $G = (V(G), E(G))$ *and* $H = (V(H), E(H))$*, are said to be* ***homomorphic*** *if there exists a mapping* $\gamma : V(G) \mapsto V(H)$ *such that if* $v_1, v_2 \in V(G)$ *and* $\gamma(v_1) = v_1' \neq v_2' = \gamma(v_2)$ *then* v_1 *and* v_2 *are adjacent if and only if* v_1' *and* v_2' *are adjacent.*

Definition 2.1.15 *A* ***automorphism*** ϕ *of a graph* G *is a isomorphic mapping from* G *to itself.*

Definition 2.1.16 *A graph* G *is* ***symmetric*** *if it has a non-trivial automorphism.*

Definition 2.1.17 *The* ***Euler*** ϕ ***function*** *of a positive integer* n*, denoted* $\phi(n)$*, is the number of positive integers* i *less than* n *such that* i *and* n *are relatively prime.*

Definition 2.1.18 *A* ***group*** *is a set* $\mathbb{G}$ *together with a binary operation* $\otimes$ *that satisfies the 4 properties:*

(a) $\mathbb{G}$ *is closed under the operation* $\otimes$*, meaning that if* x, y *are in* $\mathbb{G}$ *then so is* $x \otimes y$.

(b) The operation $\otimes$ is associative, that is, $(x \otimes y) \otimes z = x \otimes (y \otimes z)$.

(c) There is an element in $\mathbb{G}$ called e such that $e \otimes x = x = x \otimes e$ for all $x \in \mathbb{G}$.

(d) Each element x of $\mathbb{G}$ has a so-called inverse, x^{-1}, such that $x \otimes x^{-1} = e = x^{-1} \otimes x$.

Note 2.1.19 *If in addition a group has the property that $x \otimes y = y \otimes x$ for all $x, y \in \mathbb{G}$, then the group is called an Abelian group. We will only deal with finite Abelian groups in this thesis.*

Definition 2.1.20 *A **subgroup** $\mathbb{H}$ of a group $\mathbb{G}$ is a non-empty subset of $\mathbb{G}$ which is a group under the same binary operation as $\mathbb{G}$.*

Definition 2.1.21 *A **coset** of a subgroup $\mathbb{H}$ of a group $\mathbb{G}$, denoted $g \otimes \mathbb{H}$, is the set $\{g \otimes h | h \in \mathbb{H}\}$.*

Definition 2.1.22 *The **subgroup generated by** $x \in \mathbb{G}$ is the set $e, x, x \otimes x, x \otimes x \otimes x, ..., x^{-1}$. This set is denoted $\langle x \rangle$.*

Note 2.1.23 *$\langle x \rangle$ is in fact a subgroup of $\mathbb{G}$.*

Definition 2.1.24 *The **order** of a group $\mathbb{G}$ is the cardinality of the set of its elements.*

2.2 Some Well Known Results

We will list some theorems that are useful in deciding whether a dense graph is Hamiltonian.

THEOREM 2.2.1 *[18] If $\delta(v) + \delta(u) \geq |V(G)|$ for every pair of distinct non-adjacent vertices $u, v \in V(G)$, then G is Hamiltonian.*

One consequence of Theorem 2.2.1 the following corollary:

Corollary 2.2.2 *[8] If G is a graph such that $\delta_{min}(G) \geq |V(G)|/2$, then G is Hamiltonian.*

THEOREM 2.2.3 *[16] If $G = (X \cup Y, E)$ is a bipartite graph with $|X| = n = |Y|$ $(n \geq 2)$ and $\delta(u) + \delta(v) \geq n + 1$ for each non-adjacent pair $u \in X$ and $v \in Y$, then G is Hamiltonian.*

THEOREM 2.2.4 *[14] If G is a d-regular 2-connected graph of order n with $d \geq |V(G)|/3$, then G is Hamiltonian.*

The following are well-known group theory results which we will utilize later.

THEOREM 2.2.5 *Let $\mathbb{H}$ be a subgroup of the group $\mathbb{G}$. Then any two cosets $g_1 \otimes \mathbb{H}$, $g_2 \otimes \mathbb{H}$ are either disjoint or identical and the order of any coset is equal to the order of the subgroup used to generate it.*

THEOREM 2.2.6 *(Lagrange's Theorem) If $\mathbb{H}$ is a subgroup of $\mathbb{G}$ then $|\mathbb{H}|$ divides $|\mathbb{G}|$.*

THEOREM 2.2.7 *(First Sylow Theorem) Let p be prime and let $\mathbb{H}$ be a group. If p^m is the highest power of p which is a factor of $|\mathbb{H}|$, then $\mathbb{H}$ has a subgroup of order p^m.*

THEOREM 2.2.8 *(Third Sylow Theorem) Let p,m and $\mathbb{H}$ be the same as in the* *pervious theorem. Then the number of subgroups of* $\mathbb{H}$*, which have order* p^m *is congruent to 1 modulo* p*.*

Chapter 3

Hamiltonian Cycles in Symmetric Graphs

Graphs are often represented as "connect the dots" diagrams were each dot represents a vertex and two dots are connected by a line if the two vertices they represent are adjacent. Because we define graphs in terms of sets and often work with graphs in terms of dots and lines they tend to be thought of as both set theoretic and geometric objects. This sometimes leads to two words, one set theoretic and one geometric in origin being used to describe the same thing. For example; symmetries and automorphisms are two words we use to describe the same thing. A graph G is said to be **vertex-transitive** if for every distinct pair of vertices $v_1, v_2 \in V(G)$ there exists an automorphism of G which maps v_1 to v_2. We can intuitively think of a symmetry of a graph as a rearrangement of that graph's vertices which preserves edges just like in a geometric setting. A vertex-transitive graph can then be thought of as "as symmetric as possible", because each pair of vertices can be thought of as "the same" under one of the symmetries (automorphisms) of that graph. It is known that not all vertex-transitive graphs are Hamiltonian. There are 6 known non-Hamiltonian connected vertex-transitive graphs (pictured below).

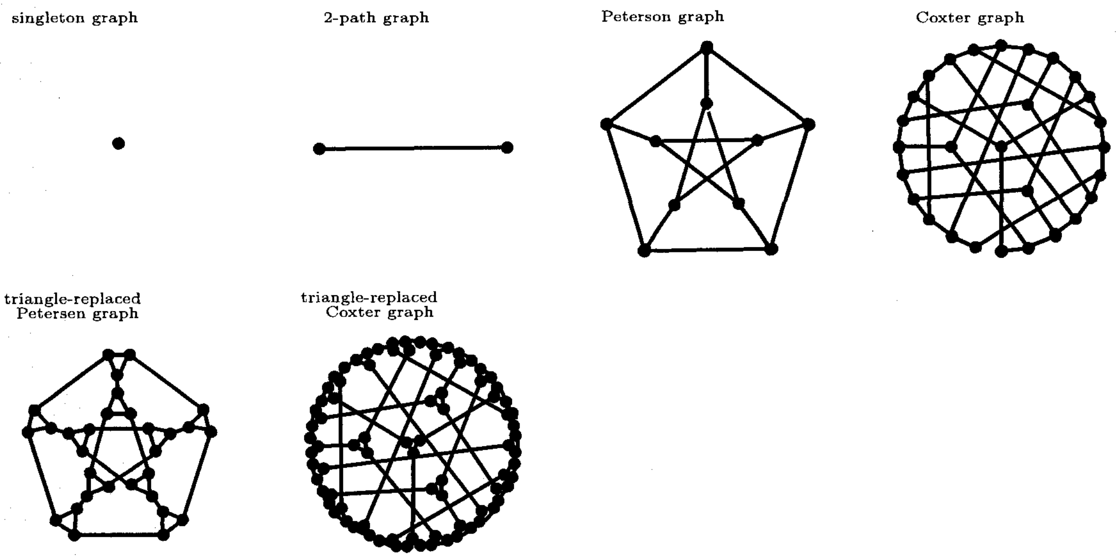

singleton graph
2-path graph
Peterson graph
Coxter graph
triangle-replaced Petersen graph
triangle-replaced Coxter graph

Figure 3.1:

Arguably, the most important conjecture relating to Hamiltonian Cycles in symmetric graphs is the following, due to Lovász [17] .

Conjecture 3.1.9 *a) Every connected vertex-transitive graph has a Hamiltonian path.*

b) Aside from the six known counterexamples, all connected vertex-transitive graphs *are Hamiltonian.*

Part b) of conjecture 3.1.9 encompasses the work done in this thesis since the graphs we will examine are vertex-transitive.

3.2 The Middle Levels Problem

One problem which Lovász's conjecture encompasses is known as the Middle Levels problem. It was first proposed in a paper of Havel's in 1982 [12] although various authors have falsely attributed the problem to authors other than Havel (most notably to: Edrös, Kelly, Déjter, Trotter). The following is Edrös's proposal of the problem:

Suppose that there is a hotel with $k+1$ people in the lobby and k people outside. There is a single door which leads from the outside to the inside and only one person can enter or exit at a time. There are $2\binom{2k+1}{k}$ ways of placing each of the $2k+1$ people either in or out of the hotel such that either k or $k+1$ end up inside. Is it always possible to go through all of these arrangements exactly once and end up back at the start by letting one person at a time in or out of the hotel?

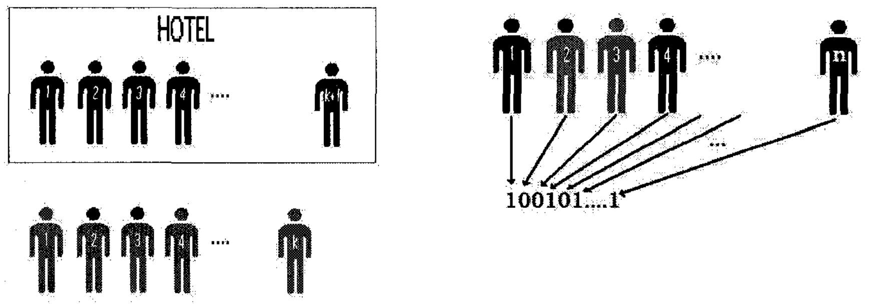


Figure 3.2:

The above interpretation of the Middle Levels problem is useful for explaining the problem, however, this formulation is difficult to use in analyzing the problem. So let us change how the problem is posed. Assign the $2k+1$ people some arbitrary order. Assign the i^{th} person the i^{th} bit in a $2k+1$ bit string and make that bit 1 if i^{th} person is inside the hotel and 0 otherwise. It is now easy to see that our problem is equivalent to the following:

For a fixed k, let X be the set of all $2k+1$ bit binary strings containing exactly k 1's and let Y be the set of all $2k+1$ bit binary strings containing exactly $k+1$ 1's. Now let M_k be the the graph with vertex set $X \cup Y$ and $x \in X$ adjacent to $y \in Y$ if and only if x differs from y in exactly one bit. Is M_k Hamiltonian for all k?

This problem has become known as the "Middle Levels Problem" because M_k can be thought of as the subgraph induced by the two middle levels of the boolean lattice.

It is conjectured that M_k is Hamiltonian for all $k > 0$. Savage, Shields and Shields [25] showed that M_k is Hamiltonian for $1 \leq k \leq 17$. Although many people have worked on this problem, there have been no published thorough explorations of the properties of M_k and related graphs. We will explore some of these properties and use them to propose improvements to the algorithm used in [25]. Our focus will be the case where $2k+1$ and k are both prime.

3.3 Definitions

For convenience, we will set $n = 2k+1$ for the remainder of this thesis. Also we will take all subscripts and superscripts to be modulo n.

Definition 3.3.1 *Given $A = \langle a_1, a_2, ..., a_n\rangle \in V(M_k)$ we call the* ***circular shift*** *of that vertex, denoted $sh(A)$, the vertex $\langle a_n, a_1, a_2, ..., a_{n-1}\rangle$. The j^{th} circular shift of A is denoted $sh^j(A) = \underbrace{sh(sh(.......sh}_{j\ times}(A))$. We also define $sh^0(A) = A$.*

Note 3.3.2 *If we are given an $A = \langle a_1, a_2, ..., a_n\rangle \in M_k$, then*

$$sh^j(A) = \langle a_{1-j}, a_{2-j}, ..., a_{n-j}\rangle$$

.

Definition 3.3.3 *Given $A = \langle a_1, a_2, ..., a_n\rangle \in V(M_k)$, we call the* ***flip*** *of that vertex, denoted $fl(A)$, the vertex $\langle a_n, a_{n-1}, a_{n-2}, ..., a_1\rangle$.*

Definition 3.3.4 *The* ***class*** *of $A \in V(M_k)$ is the set $\{B \in V(M_k) | B = sh^j(A), 0 \leq$*

$j \leq n-1\}$. *If* $A, B \in V(M_k)$ *with* $sh^m(A) = B$ *for some* m*, then we say that* B *is in the class of* A*, denoted* $B \in cl(A)$*, or, equivalently, that* A *and* B *are in the same class.*

Definition 3.3.5 *Given* $A = \langle a_1, a_2, ..., a_n \rangle \in V(M_k)$*, we call the* ***complement*** *of* A*, denoted* A^c*, the vertex* $\langle 1-a_1, 1-a_2, ..., 1-a_n \rangle$.

Note 3.3.6 *Each class has exactly n vertices in it, they are disjoint, and every class has a unique complement class.*

Definition 3.3.7 *We call a vertex* $A = \langle a_1, a_2, ..., a_n \rangle \in V(M_k)$ *an* ***l-vertex*** *if* A *is adjacent to a vertex in the class of* A^c.

Definition 3.3.8 *We call* $cl(A)$ *an l-class if* A *is an l-vertex. See Remark 3.4.3 for more details.*

Example 3.3.9 1010100 *is an l-vertex because it is adjacent to* $sh((1010100)^c) =$ 1010101.

Definition 3.3.10 *A bit string,* $A = \langle a_1, a_2, ..., a_n \rangle$*, is a* ***palindrome*** *if* $fl(A) = A$.

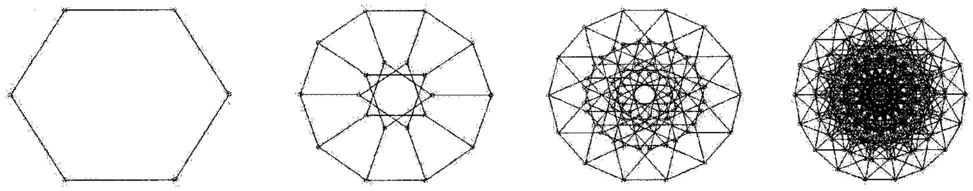
Example 3.3.11 11001010011 *is a palindrome.*

Definition 3.3.12 *A bit string,* $A = \langle a_1, a_2, ..., a_n \rangle$*, is an* ***anti-palindrome*** *if* $fl(A^c)$ *differs from* A *only in the* $(k+1)^{th}$ *position.*

Example 3.3.13 110100100 *is an anti-palindrome.*

Note 3.3.14 *The concepts 'class' and 'l-vertices' were used in previous work on the middle levels problem, specifically [4], [5], [22], [23], [24] and [25], but were given* *no formal definitions or names. However, these concepts were utilized inside proofs. Since we will be referring to these concepts throughout this thesis it was necessary to formalize them.*

3.4 Properties of M_k



The above four diagrams represent the first four M_k's. The graph on the left, M_1, is isomorphic to C_6, so the graph is itself a cycle and hence clearly Hamiltonian. However, the number of vertices in M_k grows exponentially and finding a Hamiltonian cycle becomes progressively harder.

Observation 3.4.1 *If a vertex A in M_k is adjacent to B, then $sh(A)$ is adjacent to $sh(B)$ and A^c is adjacent to B^c. Furthermore, if $B_i \in cl(B_j)$, then $B_i^c \in cl(B_j^c)$ and $sh(B_i) \in cl(sh(B_j))$.*

Lemma 3.4.2 *If A is adjacent to B_1 and B_2 from the same class, then there exist $C \neq A$ such that C is in the same class as A and B_1 is adjacent to both A and C.*

Proof: Since B_1 and B_2 are in the same class, $sh^j(B_2) = B_1$ for some $1 \leq j < n$. From Observation 3.4.1, we know that $sh^j(A)$ is adjacent to $sh^j(B_1)$ and $sh^j(B_2)$.

But $sh^j(B_2) = B_1$, which means that B_1 is adjacent to $sh^j(A)$ and A. Hence we can take $C = sh^j(A)$, which completes the proof. (Note that since $1 \leq j < n$, $sh^j(A) \neq A$).$\square$

Observation 3.4.3 *From Observation 3.4.1 we can conclude that if one vertex in a class is an l-vertex, then all the vertices in that class must be l-vertices.*

THEOREM 3.4.4 *$A = \langle a_1, a_2, ..., a_n \rangle$ is an l-vertex if and only if there exist j and s such that $a_i = 1 - a_{i-j}$ for all $i \in \{1, 2, ..n\} \backslash \{s\}$ and $a_s = a_{s-j}$.*

Proof: Assume $A = \langle a_1, a_2, ..., a_n \rangle$ is an l-vertex. This means that there exist j, $1 \leq j < n$, such that the vertex $sh^j(A^c) = \langle a'_1, a'_2, ..., a'_n \rangle$ differs from A in exactly one position. Let that position be s. Then $a_s \neq a'_s = 1 - a_{s-j}$, which implies $a_s = a_{s-j}$. We also have $a_i = a'_i = 1 - a_{i-j}$ for $i \neq s$. So A has the desired property.

Now, assume instead that $A = \langle a_1, a_2, ..., a_n \rangle$ has the properties $a_i = 1 - a_{i-j}$ for $i \neq s$ and $a_s = a_{s-j}$ for some j and s. Without loss of generality, we assume that $s = 1$. This means that $sh^j(A^c) = \langle 1 - a_{1-j}, 1 - a_{2-j}, ..., 1 - a_{n-j} \rangle = \langle 1 - a_1, a_2, a_3, a_4, ...a_n \rangle$. Hence A differs from $sh^j(A^c)$ in only the first position, and therefore, A is an l-vertex.
$\square$

Lemma 3.4.5 *Let $A = \langle a_1, a_2, ...a_n \rangle$ be an l-vertex with j and s as in Theorem 3.4.4, that is, $a_i = 1 - a_{i-j}$ for all $i \in \{1, 2, .., n\} \backslash \{s\}$ and $a_s = a_{s-j}$. Then $gcd(n, j) = 1$.*

Proof: Suppose toward a contradiction that this is not the case and $gcd(n, j) > 1$. $\mathbb{Z}_n$

is a group under addition. Since $gcd(n, j) \neq 1$, the order of the subgroup generated by j is less than the order of the group, which is n. Lagrange's Theorem tells us that the order of a subgroup divides the order of the group. Since n is odd every divisor of n must also be odd so $|\langle j \rangle|$ must also be odd. Let $x + \langle j \rangle$ be a coset of $\langle j \rangle$ which does not contain s. The distinct elements of $x + \langle j \rangle$ form the set $\{x, x+j, x+2j, ...x+(|\langle j \rangle|-1)j\}$ which is the same as $\{x, x-j, x-2j, ...x-(|\langle j \rangle|-1)j\}$ modulo n. Note that $x - (|\langle j \rangle|)j \equiv_n x$. Since $s \notin x + \langle j \rangle$, Theorem 3.4.4 gives $a_x = 1 - a_{x-j} = a_{x-2j} = ... = a_{x-(|\langle j \rangle|-1)j} = 1 - a_{x-(|\langle j \rangle|)j} = 1 - a_x$, which is a contradiction since $a_x \neq 1 - a_x$. $\square$

Observation 3.4.6 *It was shown in the proof of Theorem 3.4.4 that if*

$$A = \langle a_1, a_2, ..., a_n \rangle$$

is an l-vertex then there is a pair s, j such that $a_i = 1 - a_{i-j}$ for all $i \neq s$. If we let $s' = s - j$, then we have $a_i = 1 - a_{i-j}$ for all $i \neq s' + j$. This gives us the n equations:

$$\begin{aligned}
a_1 &= 1 - a_{1-j} \\
a_2 &= 1 - a_{2-j} \\
&\vdots \\
a_{s'} &= 1 - a_{s'-j} \\
&\vdots \\
a_{s'+j-1} &= 1 - a_{s'-1} \\
a_{s'+j} &= a_{s'} \\
a_{s'+j+1} &= 1 - a_{s'+1} \\
&\vdots \\
a_{s'+2j} &= 1 - a_{s'+j} \\
&\vdots \\
a_n &= 1 - a_{n-j}.
\end{aligned}$$

Now, the integers $0, 1, 2, .., n-1$ *form a complete system of residues modulo* n. *Since Lemma 3.4.5 tells us that* $gcd(n, j) = 1$, *the integers* $s', s'-j, s'-2j, .., s'-(n-1)j$ *also form a complete system of residues modulo* n. *Therefore,* $\{s', s'-j, s'-2j, .., s'-(n-1)j\} \equiv_n \{0, 1, 2, .., n-1\}$. *Also, observe that* $s'-(n-1)j \equiv s'+j \,(mod\, n)$, *so we can write* $a_{s'} = 1 - a_{s'-j} = a_{s'-2j} = 1 - a_{s'-3j} = ... = a_{s'-(n-1)j} = a_{s'+j}$. *Since* $s', s'-j, s'-2j, .., s'-(n-1)j$ *is a complete system of residues modulo* n, *all the bits of* A *are represented in the above string of equalities. In addition, if we know whether* $a_{s'}$ *is equal to 0 or 1, then we can use the values* s', j *and* $a_{s'}$ *to construct the bitstring that represents* A. *So* A *is completely determined by the triple* $(s', j, a_{s'})$. *Now, notice that the vertex determined by the triple* $(s'+1, j, a_{s'+1} = a_{s'})$ *is* $sh(A)$. *This means that all the vertices in the same class as* A *are determined by* $(q+s', j, a_{q+s'} = a_{s'})$

with $1 \leq q \leq n$. *Also, the complement class can be determined by triples of the form* $(q+s', j, a_{q+s'} = 1 - a_{s'})$ *with* $1 \leq q \leq n$.

Lemma 3.4.7 *If A is an l-vertex, then some vertex in $cl(A)$ is a palindrome.*

Proof: Assume that A is an l-vertex determined by the triple (s, j, a_s). Then

$$a_s = 1 - a_{s-j} = a_{s-2j} = 1 - a_{s-3j} = \ldots\ldots = a_{s+3j} = 1 - a_{s+2j} = a_{s+j},$$

which gives

$$\begin{aligned} a_s &= a_{s+j} \\ a_{s-j} &= a_{s+2j} \\ a_{s-2j} &= a_{s+3j} \\ &\vdots \\ a_{s-(k-1)j} &= a_{s+kj} \\ a_{s-kj} &= a_{s+(k+1)j}. \end{aligned}$$

Note that $s + (k+1)j \equiv_n s - kj$. Now we can rearrange the above equations to get:

$$a_{s+kj} = a_{s+kj}$$

$$a_{s+(k-1)j} = a_{s+(k+1)j}$$

$$a_{s+(k-2)j} = a_{s+(k+2)j}$$

$$\vdots$$

$$a_s = a_{s+j}.$$

Observe that, $a_{s+(k-i)j} = a_{s+(k+i)j}$ for all $1 \leq i \leq k$. There exists some circular shift of A' of A such that the k^{th} bit of A' is a_{s+kj}. The $(k+ij)^{th}$ and $(k-ij)^{th}$ bits of A' are $a_{s+(k+i)j}$ and $a_{s+(k-i)j}$, respectively. Since $k-(i)j \equiv_n n-(k+(i)j)$, A' is a palindrome, as desired. □

Lemma 3.4.8 *If A is an l-vertex, then some vertex in $cl(A)$ is an anti-palindrome.*

Proof: Let A be an l-vertex determined by the triple (s, j, a_s). The vertex $B = \langle b_1, b_2, .., b_n \rangle$, determined by the triple $(k+1, j, b_{k+1} = a_s)$, is in $cl(A)$. We have:

$$b_{k+1} = 1 - b_{k+1-j} = b_{k+1-2j} = 1 - b_{k+1-3j} = \ldots\ldots = b_{k+1+3j} = 1 - b_{k+1+2j} = b_{k+1+j}.$$

This gives us:

$$\begin{aligned}
b_{k+1} &= b_{k+1} \\
1 - b_{k+1-j} &= b_{k+1+j} \\
1 - b_{k+1-2j} &= b_{k+1+2j} \\
1 - b_{k+1-3j} &= b_{k+1+3j} \\
&\vdots \\
1 - b_{k+1-(k)j} &= b_{k+1+kj}.
\end{aligned}$$

Hence $1-b_{k+1-ij} = b_{k+1+ij}$ for all $1 \leq i \leq k$. Notice that $k+1-ij \equiv_n n-(k+1+ij)$. This means that $1 - b_{-i} = b_i$ when $i \neq k+1$, which implies that B is an anti-palindrome, as desired. □

Observation 3.4.9 *It may seem that Lemmas 3.4.8 and 3.4.7 provide two more useful characterizations of l-vertices, however, since both these proofs are not 'if and only if' proofs it might happen that some non-l-vertices are palindromes or anti-palindromes. A simple example will suffice to show that this, in fact, is the case. It is easy to see that* $A = \langle 1,1,0,0,1,0,1,1,0,1,1,0,0 \rangle$ *is an anti-palindrome. The following are all the bit strings in the inverse class of A:*

string in $cl(A^c)$	bits different from the bits of A
$\langle 0,0,1,1,0,1,0,0,1,0,0,1,1 \rangle$	13
$\langle 1,0,0,1,1,0,1,0,0,1,0,0,1 \rangle$	5
$\langle 1,1,0,0,1,1,0,1,0,0,1,0,0 \rangle$	2
$\langle 0,1,1,0,0,1,1,0,1,0,0,1,0 \rangle$	9
$\langle 0,0,1,1,0,0,1,1,0,1,0,0,1 \rangle$	7
$\langle 1,0,0,1,1,0,0,1,1,0,1,0,0 \rangle$	5
$\langle 0,1,0,0,1,1,0,0,1,1,0,1,0 \rangle$	7
$\langle 0,0,1,0,0,1,1,0,0,1,1,0,1 \rangle$	7
$\langle 1,0,0,1,0,0,1,1,0,0,1,1,0 \rangle$	5
$\langle 0,1,0,0,1,0,0,1,1,0,0,1,1 \rangle$	7
$\langle 1,0,1,0,0,1,0,0,1,1,0,0,1 \rangle$	9
$\langle 1,1,0,1,0,0,1,0,0,1,1,0,0 \rangle$	3
$\langle 0,1,1,0,1,0,0,1,0,0,1,1,0 \rangle$	5

Table 3.1:

Clearly, no bitstring in $cl(A^c)$ differs from A in exactly one bit, and therefore, A is not an l-vertex. A similar example can be constructed to show that there exist palindromes which are not l-vertices. However, not all is lost, as can be seen from the next Lemma. This Lemma will be used in the proof of Theorem 3.4.11, which is another useful characterization of l-vertices.

Lemma 3.4.10 *If some vertex in $cl(A)$ is an anti-palindrome and some other vertex in $cl(A)$ is a palindrome, then A is an l-vertex.*

Proof: Without loss of generality we can assume that $A = \langle a_1, a_2, ...a_n\rangle$ is a palindrome and that $sh^j(A)$ in an anti-palindrome. So $a_i = a_{n+1-i}$ for all i and $a_{i-j} = 1 - a_{n+1-i+j}$ for $i - j \not\equiv_n k+1$. We have:

$$\begin{aligned} a_{i-j} &= 1 - a_{n+1-i+j} \quad \text{for } i-j \not\equiv_n k+1, \\ a_i &= 1 - a_{n+1-i+2j} \quad \text{for } i \not\equiv_n k+1+j, \\ a_{n-i} &= 1 - a_{n+1-i+2j} \quad \text{for } i \not\equiv_n k+1+j. \end{aligned}$$

Now, if we let $x = n - i$ and $2j + 1 = -y$, we see that $a_x = a_{x-y}$ for $x \not\equiv_n k - j$. Hence A is an l-vertex, by Theorem 3.4.4. □

THEOREM 3.4.11 *A is an l-vertex if and only if some vertex in $cl(A)$ is an anti-palindrome and some other vertex in $cl(A)$ is a palindrome.*

Proof: Follows from Lemmas 3.4.7, 3.4.8 and 3.4.10. □

The following is equivalent to a Lemma that can be found in [4].

Lemma 3.4.12 *The number of l-classes in M_k is equal to $\phi(n)/2$.*

Proof: Let $A = (q, j, x) = \langle a_1, a_2, ..., a_n\rangle$ and let $B = (q + j, -j, x) = \langle b_1, b_2, ..., b_n\rangle$. We claim that $A = B$. We know

$$\begin{aligned} a_q &= 1 - a_{q-j} = a_{q-2j} = 1 - a_{q-3j} = ... = a_{q+j}, \\ b_{q+j} &= 1 - a_{q+2j} = b_{q+3j} = 1 - b_{q+4j} = ... = b_q. \end{aligned}$$

Now, if we write the string of equalities backwards, we get $b_q = 1 - b_{q-j} = b_{q-2j} = 1 - b_{q-3j} = ... = b_{q+j}$. Notice that $b_q = b_{q+j} = x = a_q$. Hence the two strings of equalities:

$$a_q = 1 - a_{q-j} = a_{q-2j} = 1 - a_{q-3j} = ... = a_{q+j}$$
$$b_q = 1 - b_{q-j} = b_{q-2j} = 1 - b_{q-3j} = ... = b_{q+j}$$

are identical. Thus $A = (q, j, x) = (q + j, -j, x) = B$. This also implies that the set of triples $\{(q, j, a) : q \in [n]\}$ represents the same class as the set of triples $\{(q, n-j, a) : q \in [n]\}$. Note that if $a_i = 1$, then the triple(i, j, a_i) represents a vertex in Y, and if $a = 0$, then it represents a vertex in X.

Claim: There are exactly two triples that represent each l-vertex. By proving the pervious claim we showed that each representation (q, j, x) can be paired with $(q + j, -j, x)$ which represents the same vertex. If we assume that some l-vertex $B = \langle b_1, b_2, ..., b_n\rangle$ does not have exactly two triples that represent it, then, without loss of generality, B can be represented by any of the four triples $(1, j, 1), (1 + j, n - j, 1)$ and $(q, c, 1), (q + c, n - c, 1)$ with $c \not\equiv_n j, n - j$. This means that:

$$\begin{cases} b_1 = 1 \\ b_i = 1 - b_{i-j} \quad for\ i \neq 1 - j \end{cases} \text{and} \begin{cases} b_q = 1 \\ b_i = 1 - b_{i-c} \quad for\ i \neq q - c \end{cases}$$

Hence for $i \neq q-c, 1-j$, we have $b_i = 1-b_{i-c} = 1-b_{i-j}$, which implies that $b_{i-c} = b_{i-j}$ for $i \neq q - c, 1 - j$. This then implies that $b_i = b_{i+c-j}$ for $i \neq q, 1 + c - j$. Consider the sequence $b_1, b_{1+c-j}, b_{1+2(c-j)}, ..., b_{1+(n-1)(c-j)}$. Note that because $b_i = b_{i+j-c}$ for $i \neq q, 1 + c - j$, the sequence will be 1,1,...1,0,0,...0, where the first zero will occur at

$b_{q+(j-c)}$. We know that B has $k+1$ 1's and k 0's. So it must be that our sequence 1,1,...1,0,0,...0 has $k+1$ ones followed by k zeros. This gives us $1+(k+1)(j-c) \equiv q+j-c \,(mod\, n)$. Similarly, if we instead look at $b_q, b_{q+c-j}, b_{q+2(c-j)}, ..., b_{q+(n-1)(c-j)}$ we get that $q+k(j-c) \equiv 1 \,(mod\, n)$. Combining these two results we obtain:

$$q+1+(n)(j-c) \equiv_n 1+j-c+q$$

$$q+1 \equiv_n 1+j-c+q$$

$$0 \equiv_n j-c.$$

Now, $0 \equiv_n j-c$ implies $j \equiv_n c$, which is a contradiction.

Each l-class in Y can be represented as either the set $\{(q,j,1) : q \in [n]\}$ or $\{(q, n-j, 1) : q \in [n]\}$ for some j and each l-class in X can be represented as either the set $\{(q,j,0) : q \in [n]\}$ or $\{(q, n-j, 0) : q \in [n]\}$ for some j. Hence the number of l-classes is equal to twice the number of unordered pairs $(j, n-j)$ with $gcd(j,n) = 1$ and $j < n-j$. This is exactly $\phi(n)/2$. □

THEOREM 3.4.13 *If A is adjacent to two distinct vertices from the same class, then that class must be the class containing A^c.*

Proof: Suppose that $A = \langle a_1, a_2, ..., a_n \rangle$ is adjacent to B and $sh^j(B)$ with $1 \leq j \leq k$. From the proof of Lemma 3.4.2 we know that $B = \langle b_1, b_2, ..., b_n \rangle$ is adjacent to A and $sh^j(A)$. Without loss of generality, let B differ from A in the 1^{st} bit and from $sh^j(A)$ in the t^{th} bit. Since B differs from A only in the 1^{st} bit, we have

$$\begin{cases} a_i = b_i & for\ \ i \neq 1 \\ a_1 = 1 - b_1, \end{cases}$$

and because B differs from $sh^j(A)$ only in the t^{th} bit, we have

$$\begin{cases} a_{i+j} = b_i & for\ \ i \neq t \\ a_{t+j} = 1 - b_t. \end{cases}$$

Combining these results we get that $a_i = a_{i+j}$ for $i \neq 1, t$. Without loss of generality, we can assume that A has $k+1$ 1's. Using a similar argument to the one we used in Lemma 3.4.12, we obtain $1 + (k+1)(j) \equiv t\,(mod\,n)$ and

$$a_{1+j} = a_{1+2j} = ...a_{1+(k+1)j} = 1 - a_{1+(k+2)j} = 1 - a_{1+(k+3)j} = ... = 1 - a_1.$$

Note that this implies $a_i = 1 - a_{i+(k+1)j}$ for all $i \neq 1$. Hence A can be represented by the triple $(1, (k+1)j, 1)$. Now, repeating this argument for B, we see that B can be represented by $(t, (k)j, 0)$. Since $kj \equiv_n n - (k+1)j$, we establish that B is in the complement class of A. □

Lemma 3.4.14 *The vertex (s, j, f) is adjacent to the vertices $(s - j, j, 1 - f)$ and $(s + j, j, 1 - f)$.*

Proof: Let $A = \langle a_1, a_2, ...a_n \rangle = (s, j, f)$ and $B = \langle b_1, b_2, ...b_n \rangle = (s - j, j, 1 - f)$. We know that $f = a_s = 1 - a_{s-j} = a_{s-2j} = ... = a_{s-(n-1)j}$ and that $1 - f = b_{s-j} = 1 - b_{s-2j} = b_{s-3j} = ... = b_{s-nj}$. Note that $s - nj \equiv_n s$. From $1 - f = b_{s-j} = 1 - b_{s-2j} = b_{s-3j} = ... = b_{s-nj}$ we get that $f = 1 - b_{s-j} = b_{s-2j} = 1 - b_{s-3j} = ... = 1 - b_s$. Thus A differs from B in only the s^{th} bit. Showing that (s, j, f) is adjacent to $(s + j, j, 1 - f)$ is similar. □

If we wish to build a string which represents the l-vertex A with triple (s, j, a_s), then the 1's and 0's get added in a certain order. For example, if we wanted to build the string representing the l-vertex $A \in V(M_5)$ with triple $(1, 3, 1)$, then we would start with a blank bitstring of length 5: $(*, *, *, *, *)$. We know that $a_1 = 1$, so that would be our first bit added, giving us $(1, *, *, *, *)$. Next we know that $a_1 = 1 - a_{1-3} = 1 - a_3$, so a_3 is our second bit added, giving us $(1, *, 0, *, *)$. Now, $a_3 = 1 - a_{3-3} = 1 - a_5$, so a_5, is our third bit added, giving us $(1, *, 0, *, 1)$. We continue in this manner until we have the entire string:

$$(*, *, *, *, *) \rightarrow (1, *, *, *, *) \rightarrow (1, *, 0, *, *) \rightarrow (1, *, 0, *, 1) \rightarrow (1, 0, 0, *, 1) \rightarrow (1, 0, 0, 1, 1)$$

The above gives us the order in which the bits of an l-vertex get added to the string. This means that we can represent a non-l-vertex which is adjacent to an l-vertex by just indicating in which bit it differs from its adjacent l-vertex. We will use the notation $(s, j, a_s, [i])$, to refer to such a vertex. Here $[i]$ indicates that the i^{th} bit added is the bit in which this vertex differs from the l-vertex. Similarly, we can introduce notation for any vertex in M_k by listing the positions in which it differs from some l-vertex. Define the notation to be $(s, j, a_s, [r_1, r_2, .., r_c])$, where $[r_1, r_2, .., r_c]$ is a list of bits. For simplicity we will stipulate that $r_i \neq r_j$ when $i \neq j$, because any two strings either differ or do not differ in any given bit.

Observation 3.4.15 *In say, $(s, j, f, [u_1, u_2, ...u_r])$, note that $[u_1, u_2, ...u_r]$ is just a set of instructions that tell us which bits of (s, j, f) to change in order to get $(s, j, f, [u_1, u_2, ...u_r])$. If we change s or j, the i^{th} bit we add will still have the same value. This* *means that* $(s_1, j_1, f_1,[u_1, u_2, ...u_c])$ *is adjacent to* $(s_1, j_1, f_1,[v_1, v_2, ...v_r])$ *if and only if* $(s_2, j_2, f_2,[u_1,u_2, ...u_c])$ *is adjacent to* $(s_2, j_2, f_2,[v_1, v_2, ...v_s])$. *Also,* $(s_1, j_1, f_1,[u_1, u_2, ... u_c]) = (s_1, j_1, f_1,[v_1, v_2, ...v_r])$ *if and only if* $(s_2, j_2, f_2,[u_1, u_2, ...u_c]) = (s_2, j_2, f_2,[v_1, v_2, ... v_r])$.

Lemma 3.4.16 *Let A be the vertex represented by the triple (s, j, f). Then $(s, j, f, [1])$ and $(s, j, f, [n])$ are in the same class as A^c.*

Proof: Let $A = (s, j, f) = \langle a_1, a_2, ..., a_n\rangle$. Without loss of generality we can assume that $f = 1$. This gives us

$$\begin{cases} a_s = 1 \\ a_{s-(2i-1)j} = 1 - a_s = 0, & 1 \le i \le k, \\ a_{s-2ij} = a_s = 1, & 1 \le i \le k. \end{cases}$$

This means that for $(s, j, f, [1])$ we have

$$\begin{cases} a_s = 0 \\ a_{s-(2i-1)j} = 0, & 1 \le i \le k, \\ a_{s-2ij} = 1, & 1 \le i \le k, \end{cases}$$

which is equivalent to:

$$\begin{cases} a_{(s+j)} = 0 \\ a_{(s+j)-(2i-1)j} = 1 = 1 - a_{(s+j)}, & 1 \le i \le k, \\ a_{(s+j)-2ij} = 0 = a_{(s+j)}, & 1 \le i \le k. \end{cases}$$

meaning that $(s, j, f, [1])$ represents the same vertex as $(s + j, j, 1 - f)$, which gives us the desired result. The case $(s, j, f, [n])$ is done similarly. □

Take some vertex $A = \langle a_1, a_2, ...a_n \rangle$ and some $x \in \{2, 3, ..n-1\}$ which is relatively prime to n and form the vertex $B = \langle b_1, b_2, ...b_n \rangle$ by defining $b_i = a_{x^{-1}i}$, or equivalently $a_i = b_{xi}$, for $1 \leq i \leq n$. (Since x is relatively prime to n, x is invertible modulo n). We call this action **multiplying** and we write $mult_x(A) = B$.

Observation 3.4.17 *Let $A = \langle a_1, a_2, ..., a_n \rangle = (i, j, f)$ with n prime. We know that $a_i = 1 - a_{i-j} = a_{i-2j} = 1 - a_{i-3j} = ... = a_{i-(n-1)j}$. So when we take $mult_x(A) = A' = \langle a'_1, a'_2, ..., a'_n \rangle$, we have that $a'_{xi} = 1 - a'_{xi-xj} = a'_{xi-2xj} = 1 - a'_{xi-3xj} = ... = a'_{ix-(n-1)xj}$. This gives us $A' = (ix, jx, f)$. Similarly, we can establish that $mult_x((i, j, f[r_1, r_2, ..r_s])) = (xi, xj, f[r_1, r_2, ..r_s])$. So if $mult_x(A) = B$, then for every $A' \in cl(A)$, $mult_x(A') \in cl(B)$.*

Note 3.4.18 *The vertices A with the property that $sh^j(mult_x(A)) = A$ for some j and some $x \not\equiv_n -1, 0, 1$ will have a special role later. We will refer to these as* **central** *vertices. Note that a consequence of Observation 3.4.17 is that if A is a central vertex, then so is every other vertex in its class as well as all the vertices in its inverse class.*

THEOREM 3.4.19 *If A is a central vertex with $sh^j(mult_x(A)) = A$, then for every $y \in \langle x \rangle$ there exists j such that $sh^j(mult_y(A)) = A$.*

Proof: By the remark preceding this theorem, if A is a central vertex, then so is every other vertex in $cl(A)$. We know that $mult_x(A) \in cl(A)$. Combining these results, we get that $mult_x(mult_x(A)) = mult_{x^2}(A) \in cl(A)$. We can repeat this argument to get that $mult_{x^m}(A) \in cl(A)$ for all m. Every element in $\langle x \rangle$ can be written as x^m for some m. Hence for every $y \in \langle x \rangle$ we have $mult_y(A) \in cl(A)$. Therefore, for every $y \in \langle x \rangle$ there exists a j such that $sh^j(mult_y(A)) = A$. $\square$

Lemma 3.4.20 *For any vertex $A \in V(M_k)$, $mult_{n-1}(A) = fl(A)$.*

Proof: Let $A' = \langle a_1', a_2', ...a_n' \rangle = mult_{n-1}(A)$. Observe that $j(n-1) \equiv jn - j \equiv n - j \,(mod\, n)$. This means that $a_{n-i}' = a_i$ for all i, and therefore $mult_{n-1}(A) = fl(A)$, as desired. $\square$

Observation 3.4.21 *From observation 3.4.17, we know that $mult_x(i, j, f) = (ix, jx, f)$. We wish to show that if A is central and $x \neq n-1$, then $mult_x(A) \notin cl(A)$. If we assume toward a contradiction that $jx \equiv j \,(mod\ n)$, we get that $x \equiv_n 1$, which is a contradiction because, $x \in \{2, 3, ..n-1\}$. So $jx \not\equiv j \,(mod\ n)$. If we assume that $jx \equiv n - j \,(mod\ n)$, we get $x \equiv_n n-1$. In Lemma 3.4.20 we established that $mult_{n-1}((i, j, f[r_1, r_2, ..r_s])) = fl((i, j, f[r_1, r_2, ..r_s]))$. Therefore, there is only one multiplication of a vertex which sends that vertex to a vertex in the same class and that is the one which flips it. Another consequence of this is that if $k > 2$, then no vertex is both a central vertex and an l-vertex.*

Lemma 3.4.22 *If A is a central vertex with $sh^j(mult_x(A)) = A$, then*

$$sh^{j+1-x}(mult_x(sh(A))) = sh(A).$$

Proof: Let $A = \langle a_1, a_2, ..., a_n \rangle$ be a central vertex with $sh^j(mult_x(A)) = A$ and let

$A' = \langle a'_1, a'_2, ..., a'_n \rangle$ be the vertex $sh(A)$. We know that $a_{xi-j} = a_i$ for all i. Also, $a'_i = a_{i-1}$ for all i. So $a'_{xi-x-j+1} = a_{x(i-1)-j} = a_{i-1} = a'_i$. This gives $a'_{xi+(j-x+1)} = a'_i$ for all i, which implies that $sh^{j-x+1}(mult_x(A')) = A'$, as desired. □

Corollary 3.4.23 *If n is prime and A is a central vertex in M_k with $sh^j(mult_x(A)) = A$, then for every r there is some vertex $A' \in cl(A)$ with $sh^r(mult_x(A')) = A'$.*

Proof: Let n be a prime and A a central vertex in M_k with $sh^j(mult_x(A)) = A$ and let r be fixed. By repeatedly applying Lemma 3.4.22, we get $sh^{j+y(1-x)}(mult_x(sh^y(A))) = sh^y(A)$ for all y. Now, $j + y(1-x) \equiv_n r$ for some y. Solving for y, we get $y \equiv_n (r-j)(1-x)^{-1}$. If follows that $sh^{(r-j)(1-x)^{-1}}(A)$ has the desired property. □

Lemma 3.4.24 *If n is prime and A is adjacent to A' then $mult_x(A)$ is adjacent to $mult_x(A')$ for every $x \in \{2, 3, .., n-1\}$.*

Proof: Let $A = \langle a_1, a_2, ..., a_n \rangle$ and $A' = \langle a'_1, a'_2, ..., a'_n \rangle$ be adjacent. Then A differs from A' in one bit, say a_i. Let $x \in \{2, 3, .., n-1\}$. Since n is prime and $x \in \{2, 3, .., n-1\}$ we know that x has a multiplicative inverse modulo n, namely x^{-1}. Hence $mult_x(A) = \langle a_{1x^{-1}}, a_{2x^{-1}}, ..., a_{nx^{-1}} \rangle$ and $mult_x(A) = \langle a'_{1x^{-1}}, a'_{2x^{-1}}, ..., a'_{nx^{-1}} \rangle$. However, we know that $a_j = a'_j$ for all $j \not\equiv_n i$, so that $mult_x(A)$ and $mult_x(A')$ differ in only the ix^{th} bit and are therefore adjacent. □

Observation 3.4.25 *Lemma 3.4.24 showed us that if n is prime and A is adjacent to A' then $mult_x(A)$ is adjacent to $mult_x(A')$ for every $x \in \{2, 3, .., n-1\}$. Hence the* *mapping from $V(M_k)$ to itself, given by $mult_x$, preserves edges and it is therefore a homomorphism. Since the mapping is invertible ($mult_{x^{-1}}$ is its inverse) this mapping is in fact an automorphism.*

3.5 Properties of R_k

Déjter [4] observed a useful reduction of the the problem. The problem of finding a Hamiltonian cycle in M_k can be reduced to the problem of finding a Hamiltonian path in a smaller graph, R_k. R_k is obtained in the following way:
First we obtain the reduced graph N_k. Let each vertex in N_k represent a distinct class of M_k and let two vertices in N_k be adjacent if their corresponding classes in M_k contain vertices which are adjacent. Two vertices in N_k will be called inverses of each other if they represent classes which are inverses of one another. The reduced graph R_k is obtained by letting each vertex represent a pair of vertices which are inverses of one another in N_k. Two vertices in R_k are adjacent if they represent adjacent vertices in N_k, and if a vertex in N_k is adjacent to its inverse, the corresponding vertex will have a loop in R_k. Now, because of how we construct R_k, each vertex in it represents $2n$ vertices in two classes which are inverses of one another in M_k. We will refer to these $2n$ vertices in M_k as the **parents** of the vertex they correspond to in R_k. Notice that vertices in R_k which have loop edges are exactly the vertices whose parents are l-vertices (l for loop). Déjter showed that a Hamiltonian path in R_k with loop vertices as endpoints can be used to construct a Hamiltonian cycle in M_k.

Observation 3.5.1 *The representations for the vertices of M_k that we used earlier can be carried over to R_k; we can even further simplify the notation: For example, in $(i, j, f[u_1, u_2, ...u_s])$, the i and f are redundant, so we can represent the same vertex with $(j, [u_1, u_2, ...u_s])$. Note that the l-vertex (i, j, f) will now be represented as (j).*

The following is a special case of a Theorem proven in [4].

THEOREM 3.5.2 *If there exist a Hamiltonian path from* (1) *to* (k) *in* R_k*, then there exists a Hamiltonian cycle in* M_k*.*

Proof: Using the relationship between R_k and M_k we can turn a Hamiltonian path in R_k into $2n$ disjoint paths which together include all of the vertices of M_k.

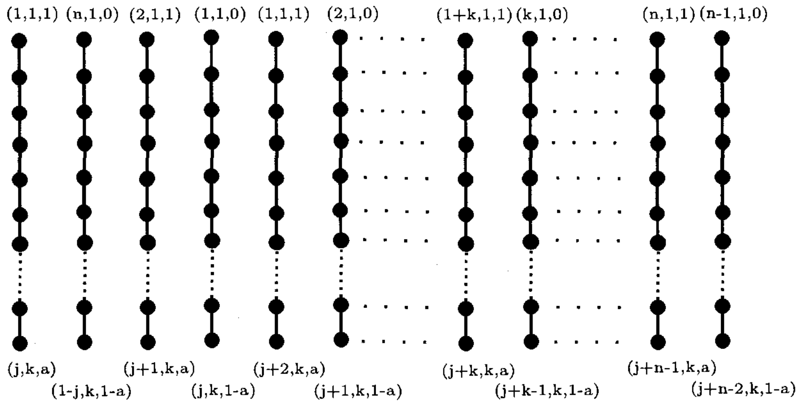


Figure 3.3:

Now, all we need to do is connect the paths into a cycle. We add the n edges of the from $(j, 1, 1)(j - 1, 1, 0)$ (we showed in Lemma 3.4.14 that these edges exist). Now we have n disjoint paths which together include all of the vertices of M_k.

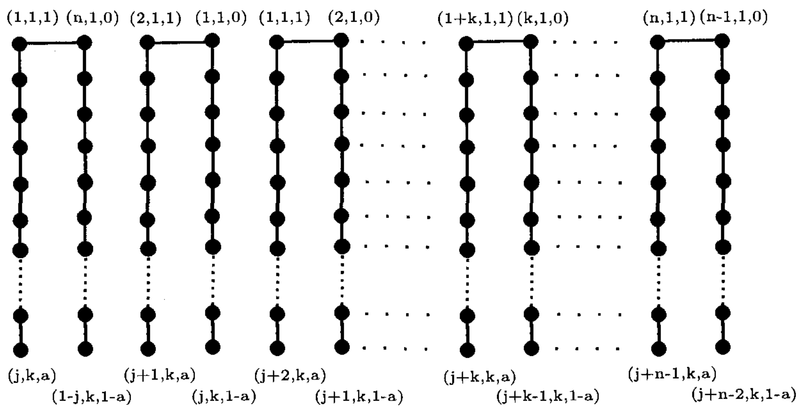

(1,1,1) (n,1,0) (2,1,1) (1,1,0) (1,1,1) (2,1,0) (1+k,1,1) (k,1,0) (n,1,1) (n-1,1,0)
(j,k,a) (1-j,k,1-a) (j+1,k,a) (j,k,1-a) (j+2,k,a) (j+1,k,1-a) (j+k,k,a) (j+k-1,k,1-a) (j+n-1,k,a) (j+n-2,k,1-a)

Figure 3.4:

We know we have n edges of the from $(j, k, 1)(j + k, k, 0)$. We claim that adding these completes the cycle.

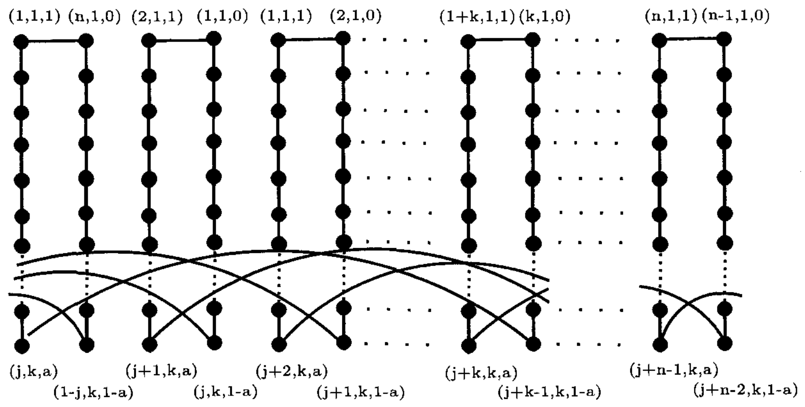

(1,1,1) (n,1,0) (2,1,1) (1,1,0) (1,1,1) (2,1,0) (1+k,1,1) (k,1,0) (n,1,1) (n-1,1,0)
(j,k,a) (1-j,k,1-a) (j+1,k,a) (j,k,1-a) (j+2,k,a) (j+1,k,1-a) (j+k,k,a) (j+k-1,k,1-a) (j+n-1,k,a) (j+n-2,k,1-a)

Figure 3.5:

It suffices to show that we can get from any vertex of the form $(j, k, 1)$ to any other vertex of that form. We know that $(j, k, 1)$ is connected through a path to $(j-1, k, 0)$

which is connected to $(j-k-1, k, 1) = (j-(k+1), k, 1)$. Repeating this we can get from $(j, k, 1)$ to $(j - x(k+1), k, 1)$ for any x. Since $gcd(k+1, n) = 1$, the integers $j, j-(k+1), j-2(k+1), ..., j-(n-1)(k+1)$ form a complete system of residues modulo n. Thus we can get from any vertex of the form $(j, k, 1)$ to any other vertex of that form. □

We now illustrate the construction in Theorem 3.5.2 by example:

Let us start with R_3 pictured below. We construct a Hamiltonian path between two loop vertices (1) = 1010101, (3) = 1111000. This path can be explicitly written as: 1010101, 1010001, 1011001, 1011000, 1111000 and is represented below as a colored path.

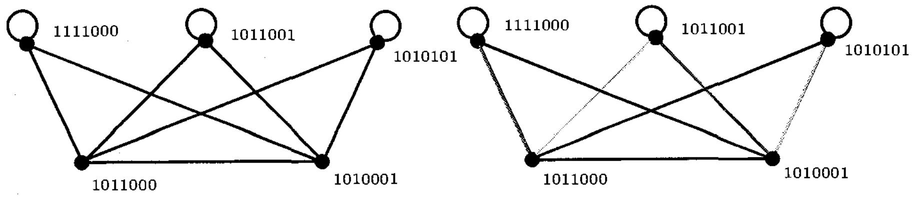


Figure 3.6:

This path in R_3 represents 2(7) = 14 disjoint paths in in M_3. The edges are color matched so one can see which edges in R_3 correspond to edges in M_3. Now add all edges of the form $(j, 1, 1)(j-1, 1, 0)$ as well as all edges of the form $(j, k, 1)(j+k, k, 0)$. This, along with the paths we already have, gives us a Hamiltonian cycle in M_k.

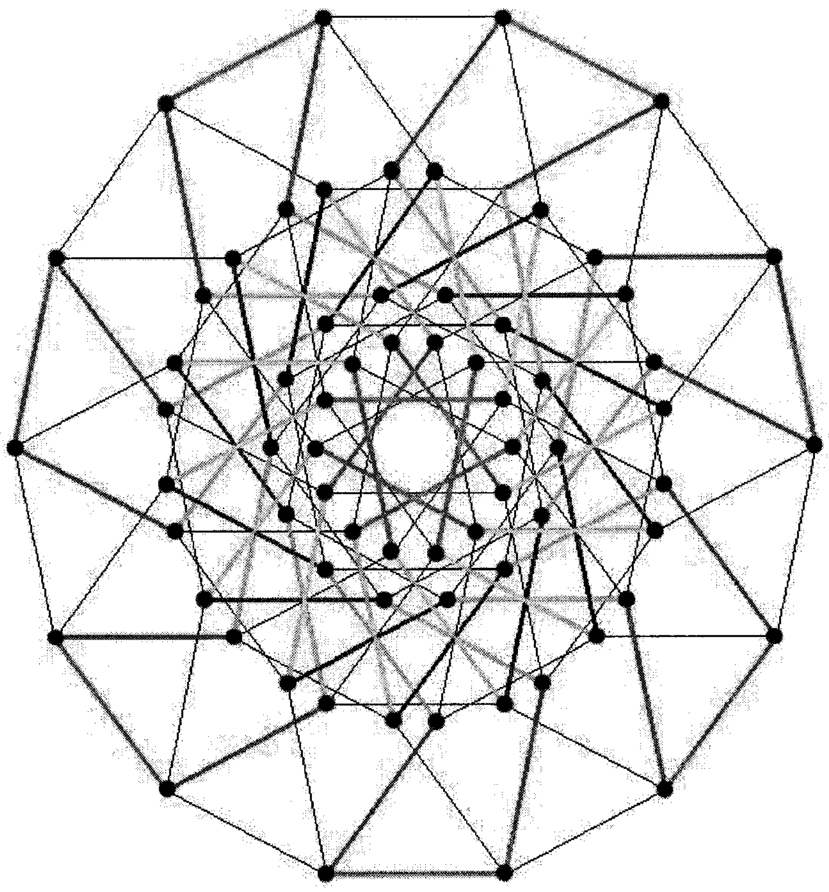


Figure 3.7:

If the reader wishes to follow the path directly, the following diagram illustrates it with the edges color matched as in the previous diagrams.

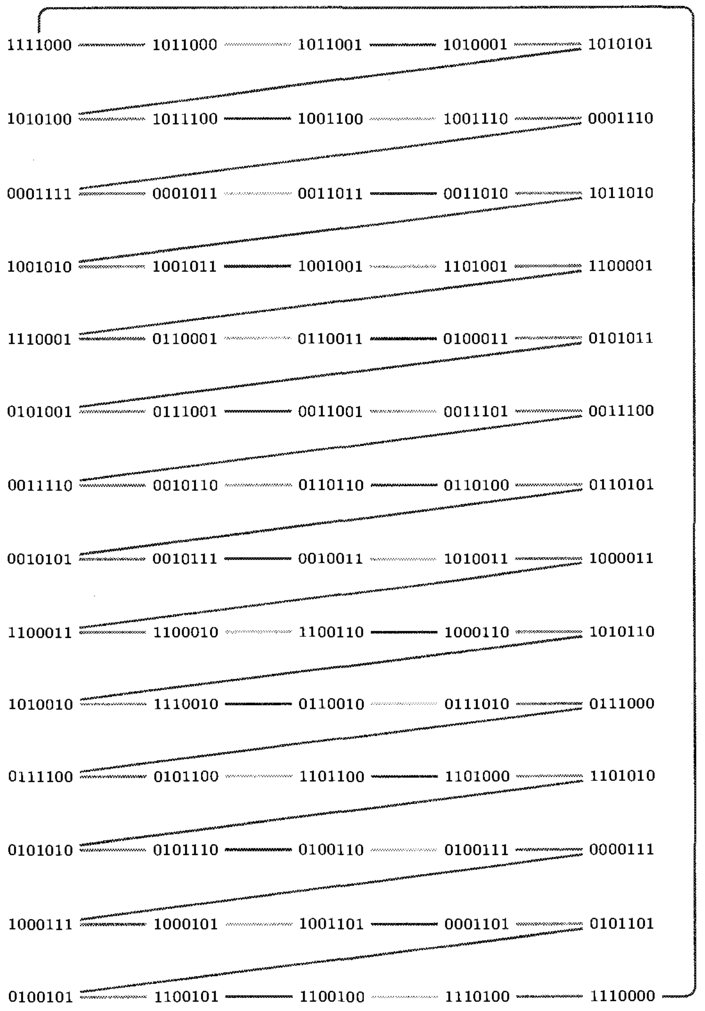


Figure 3.8:

Corollary 3.5.3 *Let n be prime. If there exists a Hamiltonian path in R_k from (a) to (b) and $gcd(a,b) = 1 = gcd(a+b,n)$, then there exists a Hamiltonian cycle in M_k.*

The proof of this result is essentially identical to the proof of Theorem 3.5.2 with a replacing 1 and b replacing k.

Lemma 3.5.4 *R_k contains $\frac{(n-1)!}{(k)!(k+1)!}$ vertices.*

Proof: Each class has n vertices in it and each class has a corresponding unique inverse class, so that each vertex in R_k represents exactly $2n$ vertices in M_k. Since there are $2\binom{n}{k}$ vertices in M_k, then the number of vertices in R_k is $2 \times \binom{n}{k}/(2n) = \frac{(n-1)!}{(k)!(k+1)!}$, as claimed.□

Dejter noted that the number of vertices in R_k is always a Catalan number. We will explore the relationship between R_k and the Catalan numbers more thoroughly in the next section.

Observation 3.5.5 *The number of loop vertices in R_k is exactly $\phi(n)/2$. This follows directly from Lemma 3.4.12.*

Observation 3.5.6 *If n is prime then, R_k has $k+1$ vertices of degree $k-1$ and $\frac{(n-1)!}{(k)!(k+1)!} - (k+1)$ vertices of degree $k+1$.*

When reducing M_k to R_k, the two most natural automorphisms, shifting and flipping the string are used to condense M_k into R_k. A natural question is whether there are any more automorphisms present in R_k, and if so, can we use them to reduce the R_k further into an even smaller graph? The answer to this question is positive in some special cases. To illustrate these, we need to develop some further machinery.

We want to extend the definition of multiplication to R_k: For $A, B \in R_k$ we say that $mult_x(A) = B$ if for every parent of A, say, $A' \in M_k$, $mult_x(A')$ is a parent of B. Using this definition we can now extend the definition of **central vertices** to include vertices in R_k. That is: $A \in R_k$ is a central vertex if $mult_x(A) = A$ for some $x \not\equiv_n -1, 0, 1$. We have already observed that if $A \in M_k$ is a central vertex, then so is every other vertex in both its class and its inverse class. This means that central vertices in R_k have parents which are central vertices in M_k.

Lemma 3.5.7 *Let $n = 2k + 1$ be prime. If a vertex $A \in R_k$ can be represented as both $(i, [u_1, u_2, ..u_s])$ and $(j, [u_1, u_2, ..u_s])$ with $i \not\equiv_n j, n - j$, then A can be represented as $(r, [u_1, u_2, ..u_s])$ for any $r \in \langle ij^{-1} \rangle$.*

Proof: This is a consequence of Note 3.4.18 and Theorem 3.4.19: If $A \in M_k$ is central, then for every $r \in \langle ij^{-1} \rangle$ there exists an m such that $sh^m(mult_r(A)) = A$. $\square$

THEOREM 3.5.8 *Let $k > 3$. If both k and $n = 2k + 1$ are prime, then there is at least one central vertex in M_k.*

Proof: The set $\mathbb{Z}_n \backslash \{0\}$ is a group under multiplication. This groups has order $2k$. By the first Sylow theorem, we know that this group has some subgroup $\mathbb{G}$ of order k. Let the the elements of $\mathbb{G}$ be $\{x_1, x_2, ...x_k\}$. Now let $A = \langle a_1, a_2, .., a_n \rangle$ with $a_i = 1$ if $i \in \{x_1, x_2, ...x_k\}$ and $a_i = 0$ otherwise (note that A has k 1's and $k + 1$ 0's as desired). We claim that A is central. Consider $A' = \langle a'_1, a'_2, .., a'_n \rangle = mult_{x_i}(A)$. The cosets of the subgroup $\mathbb{G}$ are either disjoint or identical. So $j\mathbb{G} = \mathbb{G}$ if $j \in \mathbb{G}$ and $j\mathbb{G} \cap \mathbb{G} = \emptyset$ otherwise. Hence $jx_i^{-1} \in \{x_1, x_2, ...x_k\}$ if and only if $j \in \{x_1, x_2, ...x_k\}$, since $x_i^{-1} \in \mathbb{G}$. Thus, $a'_j = a_{jx_i^{-1}} = 1$ if and only if $j \in \{x_1, x_2, ...x_k\}$, and $a'_j = 0$ otherwise. So $mult_{x_i}(A) = A$. In fact, $mult_{x_i}(A) = A$ regardless of which x_i we choose. This establishes that A is a central vertex, because at least one element of

$\mathbb{G}$ is not $-1, 0, 1$. $\square$

THEOREM 3.5.9 *If k and $n = 2k+1$ are both prime, then there are exactly two central vertices in R_k and these two vertices are adjacent to each other.*

Proof: Let A be a central vertex in R_k and $\mathbb{G}$ the group from the previous theorem. From Corollary 3.4.23 and Theorem 3.4.19, we know that for any pair $b \in \mathbb{G}$ and $c \in \{0, 1, 2, 3, ...n-1\}$ there exist a parent of A, say $A' \in M_k$, such that $sh^c(mult_b(A')) = A' = \langle a'_1, a'_2, ..a'_n \rangle$. This means that for any i, $a'_{bi-c} = a'_i$. Let $f(x) = bx - c$. We know that for any fixed i, $a'_i = a'_{f(i)} = a'_{f^2(i)} = a'_{f^3(i)} = ...$ Since $f^r(i)$ can take on a finite number of values, there exist some $0 \leq r_1 < r_2 < n$ such that $f^{r_1}(i) = f^{r_2}(i)$. Without loss of generality we can assume that r_2 is minimum. The function $f(x)$ is invertible: $f^{-1}(x) = b^{-1}x + b^{-1}c$. So if $r_1 \neq 0$:

$$\begin{aligned} f^{r_1}(i) &\equiv_n f^{r_2}(i) \\ f^{-r_1}(f^{r_1}(i)) &\equiv_n f^{-r_1}(f^{r_2}(i)) \\ i &\equiv_n f^{r_2 - r_1}(i) \end{aligned}$$

This contradicts the minimality of r_2 since $r_2 - r_1 < r_2$. So r_1 must be zero, meaning that $i \equiv_n f^{r_2}(i)$. This means $f^j(i) \equiv_n f^{r_2+j}(i)$ for all j. If $r_2 > k$ then we have more that $k+1$ bits all equal to one another. This is a contradiction, because A' has either $k+1$ 1's and k 0's or $k+1$ 0's and k 1's. So $0 < r_2 \leq k$. If we assume that $r_2 = 1$, then we can solve for i: $i \equiv_n bi - c \rightarrow i \equiv_n c(b-1)^{-1}$. Since we can solve for it, there is exactly one value of i for which r_2 is 1. If we assume that $r_2 = 2$ we get

$$\begin{aligned} i &\equiv_n f^2(i) \\ i &\equiv_n b^2 i - cb - c \\ c(b+1) &\equiv_n (b^2-1)i \\ (b^2-1)^{-1}c(b+1) &\equiv_n i \\ (b^2-1)^{-1}c(b+1) &\equiv_n i \\ (b+1)^{-1}(b-1)^{-1}c(b+1) &\equiv_n i \\ (b-1)^{-1}c &\equiv_n i \end{aligned}$$

So in this case we notice that $i \equiv_n c(b-1)^{-1}$ happens to be the same i we got when solving for i when $r_2 = 1$. Thus the minimum r is never 2 for any i. Claim: $f^m(x) = b^m x - c(b^{m-1} + b^{m-2} + .. + b + 1)$. We proceed to show this by induction. The base case, $f(x) = f^1(x) = bx + c$, is given. Now, assume that the claim holds for all $m < M - 1$:

$$\begin{aligned} f^M(x) &= f(f^M(x)) \\ f^M(x) &= f(b^{M-1}x - c(b^{M-2} + b^{M-3} + .. + b + 1) \\ f^M(x) &= b(b^{M-1}x - c(b^{M-2} + b^{M-3} + .. + b + 1) + c \\ f^M(x) &= (b^M x - c(b^{M-1} + b^{M-1} + .. + b)) + c \\ f^M(x) &= b^M x - c(b^{M-1} + b^{M-1} + .. + b + 1) \end{aligned}$$

Thus $f^m(x) = b^m x - c(b^{m-1} + b^{m-2} + .. + b + 1)$. Note that $b^{2k} \equiv_n 1$ by Fermat's Little Theorem. So we have:

$$\begin{aligned} f^{2k}(x) &= b^{2k}x - c(b^{2k-1} + b^{2k-2} + .. + b + 1) \\ &= b^{2k}x - c(b^{2k} - 1)(b-1)^{-1} \\ &\equiv_n (1)x - c(1-1)(b-1)^{-1} \\ &= x \end{aligned}$$

Thus $f^{2k}(x) = x$. This means that $r_2 | 2k$. We showed that $r_2 \neq 2$ and that $r_2 \leq k$ so r_2 can only be 1 or k since we assumed that k was prime. So r_2 is 1 when $i = c(b-1)^{-1}$, and k, otherwise. So we have 3 sets of bits of sizes 1, k and k each of which must be either all 1's or all 0's. Since there are either k or $k+1$ 1's, one of the sets of size k has to be all 1's and one of the sets has to be all 0's. If the bit in set of size one is 1, then we get one of the central vertices, and if it is 0, we get a neighboring central vertex. Observation 3.4.21 tells us that l-vertices are not central vertices. So these two adjacent central vertices must be distinct parents of two distinct central vertices in R_k. Thus we have constructed two central vertices in R_k. By Corollary 3.4.23 and Theorem 3.4.19, we know that had we had chosen any other b, c we would have gotten the same two central vertices. So they must be unique. □

3.6 The Automorphisms of R_k

Let k and n both be prime. The set $\mathbb{Z}_n \backslash \{0\}$ is a group of order $2k$ under multiplication. The first Sylow Theorem (Theorem 2.2.7) tells us that $\mathbb{Z}_n \backslash \{0\}$ has a subgroup $\mathbb{G}$ of order k. We know show that this subgroup is unique.

THEOREM 3.6.1 *If n and k are both prime then, $\mathbb{Z}_n \backslash \{0\}$ has exactly one subgroup of order k.*

Proof: We know, by the Third Sylow Theorem (Theorem 2.2.8), that the number of such subgroups is congruent to 1 modulo k. So if we assume toward at contradiction that there is more than one such subgroup, then there must be at least $k+1$ of them. At least two of these subgroups must have a common element, y, other than 1, by the pigeonhole principle. However, since k is prime, these subgroups are cyclic and hence both equal to $\langle y \rangle$, a contradiction. So $\mathbb{G}$ is unique. □

The mapping from $\phi : R_k \mapsto R_k$ given by $A \mapsto mult_x(A)$ is an automorphism for all $x \in \{1, 2, 3..., n-1\}$. Let $A = \langle a_1, a_2, .., a_n \rangle$ with $a_i = 1$ if and only if $i \in \mathbb{G}$. We showed in the proof of Theorem 3.5.8 that if $x \in \mathbb{G}$, then $A = mult_x(A)$. However, if $x \notin \mathbb{G}+\{0\}$, then $ix \in \mathbb{G}$ if and only if $i \notin \mathbb{G}+\{0\}$. This means that when $x \notin \mathbb{G}+\{0\}$ and C and C' are the two central vertices, $mult_x(C) = C'$ and $mult_x(C') = C$. This happens because $mult_x(C^c)$ differs from C in exactly the 0^{th} ($=n^{th}$) bit. Hence the mapping given by $mult_x(C)$ with $x \in \mathbb{G}$ fixes the central vertices and when $x \notin \mathbb{G} + \{0\}$ it maps them to one another. So we will refer to the k automorphisms that fix the central vertices as **central fixing automorphisms**, or CFA's for short.

Lemma 3.6.2 *If n and $k > 2$ are both prime, then the unique subgroup of $\mathbb{Z}_n\backslash\{0\}$ of order k is $\langle x^2 \rangle$, where x is an arbitrary element in $\mathbb{Z}_n\backslash\{-1, 0, 1\}$.*

Proof: The only element of order 2 in $\mathbb{Z}_n\backslash\{0\}$ is -1. Lagrange's Theorem tells us that the order of an element divides the order of the group. Thus if $x \in \mathbb{Z}_n\backslash\{-1, 0, 1\}$ then, x has either order $2k$ or k. It suffices to show that x^2 is of order k. By Ferrmat's Little Theorem $(x^2)^k \equiv_n x^{2k} \equiv_n x^{n-1} \equiv_n 1$. Hence $(x^2)^k \equiv_n 1$ and x^2 has order k. $\square$

The above Lemma shows us that, we can use any square to generate the subgroup of order k. We showed the uniqueness of this subgroup in Theorem 3.6.1. Using the proofs of Theorems 3.5.8 and 3.5.9 we can then use this subgroup to generate all central vertices in R_k.

Let's explore the automorphisms of R_3. R_3 has 5 vertices. Three of them are loop vertices and the other two are central vertices. We label the three loop vertices $(1), (2)$ and (3) and the two central vertices C_1 and C_2. The automorphism represented by the diagram is written underneath it. $mult_1$ is, of course, the identity. Since the unique subgroup of $\mathbb{Z}_7\backslash\{0\}$ of order 3 is $\{1, 2, 4\}$ $mult_1$, $mult_2$ and $mult_4$ are CFA's. If there exists some CFA that maps A to B, then we say that A and B are **compatible**. Similarly, if there exist some CFA that maps the path/cycle $A_1, A_2, .., A_m$ to $B_1, B_2, .., B_m$ we say that $A_1, A_2, .., A_m$ and $B_1, B_2, .., B_m$ are compatible. The set of all vertices which are compatible with A will be referred to as the **compatibility class** of A, denoted $com(A)$. A is compatible with itself since the identity

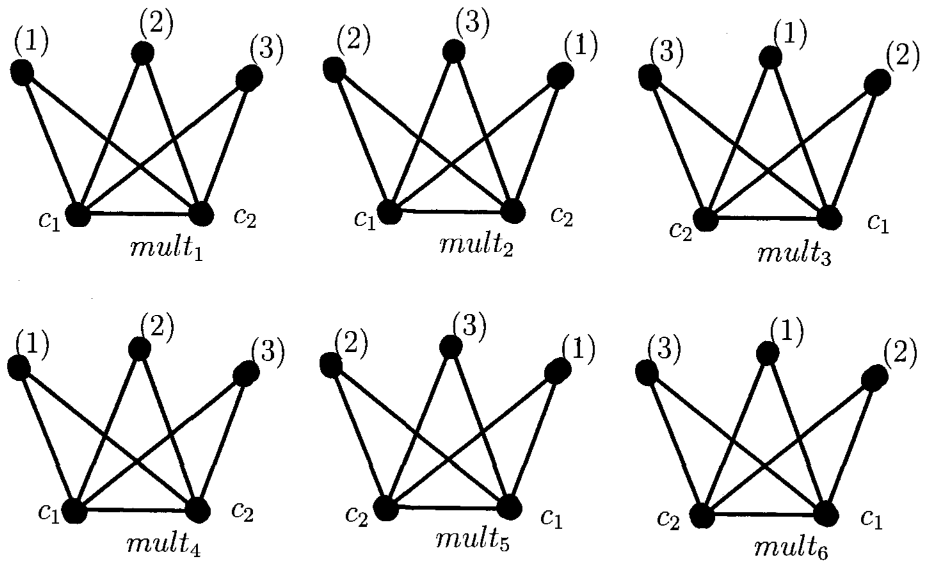


Figure 3.9:

automorphism is a CFA. If n and k are prime, all the compatibility classes, with the exception of the compatibility class that contains only the two central vertices, have k vertices in them.

3.7 Catalan Numbers

The Catalan numbers arise naturally in combinatorics in many counting problems; often these problems involve recursively defined objects. The k^{th} Catalan number, C_k, is defined as $\frac{1}{k+1}\binom{2k}{k}$ Dejter noted that the number of vertices in R_k is always a Catalan number (specifically R_k, has C_k vertices). An example of one of the ways that Catalan numbers arise is as the result of counting the number of binary strings of length $2k$ with k 1's and k 0's such that no initial segment of the string has more

0's than 1's. The number of these string is the Catalan number C_k. If we add a prefix 1 to such a string, it will have the property that every prefix will have more 1's than 0's. This leads us to the following definition:

Definition 3.7.1 *We will refer to a binary string as* ***valid*** *if every prefix of that string (including the prefix which is the entire string) has more 1's than 0's.*

Note that we can create C_k valid bit strings of length n by taking the C_k strings of length $2k$ that have k 1's and k 0's and the property that no initial segment of the string has more 0's than 1's and adding the prefix 1 to each of them.

THEOREM 3.7.2 *Among all the circular shifts of a bit string with $k+1$ 1's and k 0's there is exactly one valid bit string.*

Proof: Suppose toward a contradiction that there are two distinct valid bitstrings A and A' such that $sh^j(A) = A'$ for some j. Since $A = \langle a_1, a_2, ...a_n \rangle$ is valid $\sum_{i=1}^{s}(-1+2a_i) > 0$ for all $1 \leq s \leq n$. Specifically this means that $\sum_{i=1}^{n+1-j}(-1+2a_i) > 0$. Since $1 = \sum_{i=1}^{n+1-j}(-1+2a_i) + \sum_{i=n+1-j}^{n}(-1+2a_i)$, we know that $\sum_{i=n+1-j}^{n}(-1+2a_i) \leq 0$. But $sh^j(A) = A'$ which means that $A' = \langle a_{1-j}, a_{2-j}, ..., a_{n-j} \rangle = \langle a'_1, a'_2, ..., a'_n \rangle$. This means that $0 \leq \sum_{i=n+1-j}^{n}(-1+2a_i) = \sum_{i=1}^{j}(-1+2a'_i)$ and hence the prefix $a'_1, a'_2, ..., a'_{n-j}$, of A' does not contain more 0's than 1's, a contradiction.
Therefore, among all the circular shifts of of a bit string with $k+1$ 1's and k 0's, there is at most 1 valid bitstring. However, we can create C_k valid strings with $k+1$ 1's and k 0's (as noted above) and there are C_k classes of circular shifts. Hence there is exactly one valid bit string among the circular shifts of a bit string with $k+1$ 1's and k 0's. □

The above theorem tells us that every vertex in R_k has a unique representation as a valid string. This is useful in both building and searching through R_k. All valid strings of length n can be generated recursively [27]. However, for the purpose of out experiments, we choose to generate them non-recursively.

We know that the first two bits of a valid bit string must both be 1. We will start there and sequentially add bits to the end of the existing string. Let us say we have some partially completed string $A = \langle a_1 = 1, a_2 = 1, a_3, ..., a_m \rangle$. We then count the number of ones in the partially completed string of length m. If this number is $k+1$, we add $n - m$ 0's to the end of the string and we have a valid string of length n. If this is not the case, we certainly have less then $k + 1$ 1's, and then we will compute the sum $S = \sum_{i=1}^{m}(-1+2a_i)$. If this sum is greater than 1, we replace A with the two partially completed strings $\langle a_1 = 1, a_2 = 1, a_3, ..., a_m, a_{m+1} = 0 \rangle$ and $\langle a_1 = 1, a_2 = 1, a_3, ..., a_m, a_{m+1} = 1 \rangle$. If the sum is 1, then adding a 0 to the end of the string would make it non-valid, so we replace A with $\langle a_1 = 1, a_2 = 1, a_3, ..., a_m, a_{m+1} = 1 \rangle$. If we continue extending the partial strings in this way we will end up with a list of all valid string of length n.

Having a unique representation for each vertex of R_k allows us to construct the edges of R_k easily. Starting with a valid string $A = \langle a_1, a_2, ...a_n \rangle$, for each bit a_i with $a_i = 1$, we flip every bit except a_i to get a bit string B, and then, if B is not valid, we can simply look through the shifts of B until we find one that is. If we do this for all possible a_i's we will have all the neighbors of A. This is quite useful, because we can now construct R_k directly without needing to construct M_k first.

Chapter 4

Reducing the Middle Levels Problem

4.1 A Reduction of the Middle Levels Problem

In the previous sections we developed the machinery that we will use in this chapter to propose a method different than those used by previous authors for determining whether a particular graph M_k is Hamiltonian. We believe that this method has the potential to verify that a particular M_k is Hamiltonian in a more computationally efficient manner. Much like previous methods, our method involves a Hamiltonian cycle/path heuristic. The difference lies in how the problem is broken down into steps.

In Observation 3.4.25, we justified that the reduced graph R_k has non-trivial automorphisms. We will utilize these automorphisms in order to create a graph which has fewer vertices than R_k. This is done in a way similar to the way R_k is constructed from M_k (discussed earlier), with one major difference. In R_k, every vertex, represents $2n$ vertices from M_k. In this new reduced graph, which we will refer to as Z_k, each vertex represents k vertices in R_k, but there are two vertices in R_k, which are not represented in Z_k. These are the two central vertices whose existence was shown in Theorem 3.5.9.

There are 3 major steps in our method:

1) We construct Z_k for the particular R_k.

2) We use a Hamiltonian path heuristic to find a Hamiltonian cycle in Z_k. This cycle corresponds to k disjoint cycles in R_k.

3) We interconnect the cycles as well as the two central vertices into one Hamiltonian path by removing one edge from each cycle and interconnecting them using edges in R_k. The Hamiltonian cycle in step 3 corresponds to a Hamiltonian cycle in M_k.

We would like to remind the readers that our explorations are only for the case where both n and k are prime. If this is not the case, we have no guarantee that the number of central vertices is 2, or that each compatibility class has k vertices in it. For example, R_6 has 6 central vertices.

We illustrate this method for the case $k = 5$ and $n = 11$. (Note that these are both primes):

We start off with R_5 which is pictured below. The two central vertices are placed over top of each other to de-clutter the image. They are represented as a star and the 5 central vertices are represented as circles. All other vertices are represented with points.

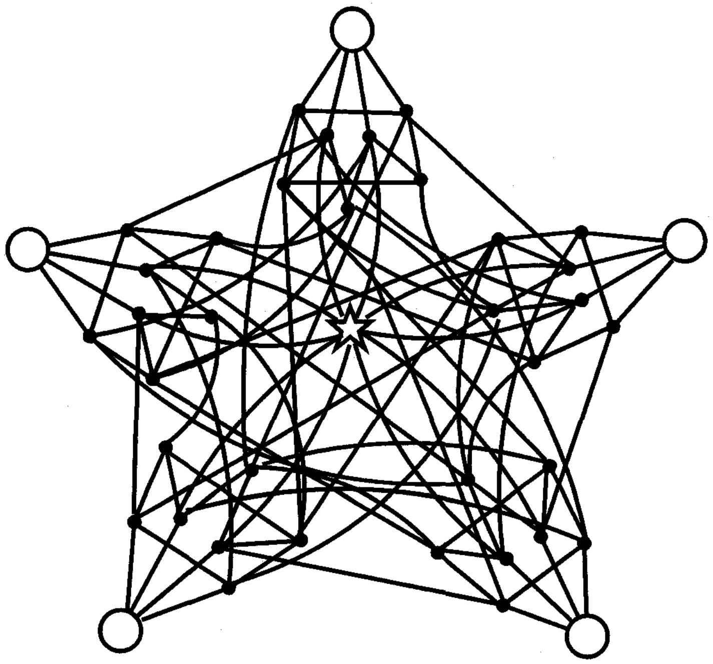


Figure 4.1:

Now we find Z_5, which is isomorphic to 5 disjoint subgraphs of R_5.

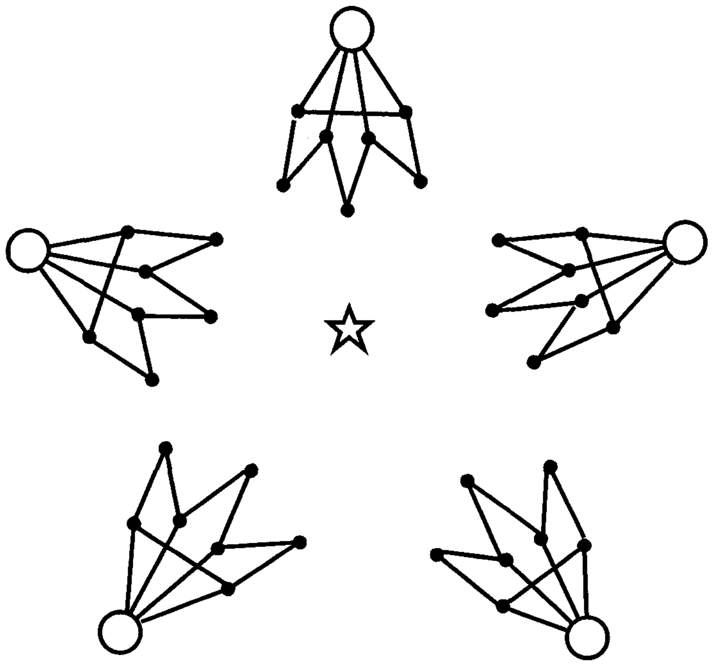


Figure 4.2:

We now find a Hamiltonian cycle in Z_5 which corresponds to 5 cycles in R_k.

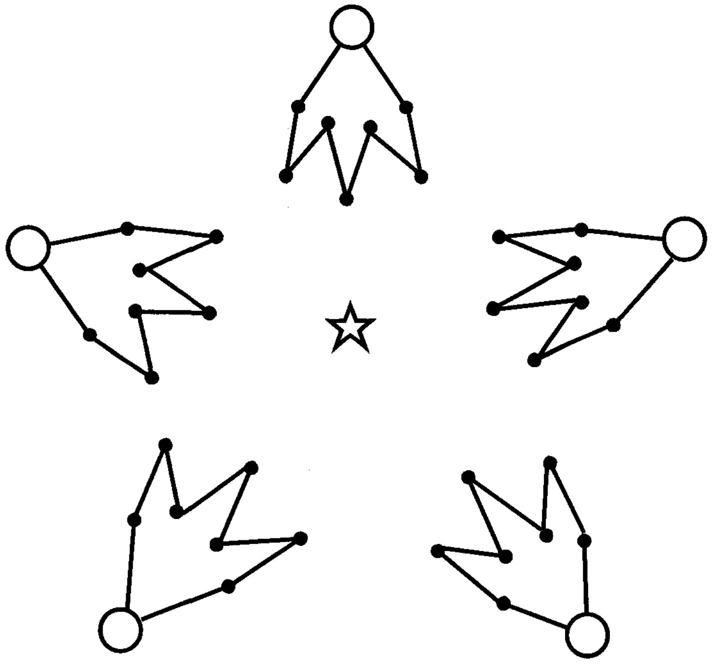


Figure 4.3:

Breaking a single edge in each cycle, we interconnect the broken cycles into a long Hamiltonian path in R_k.

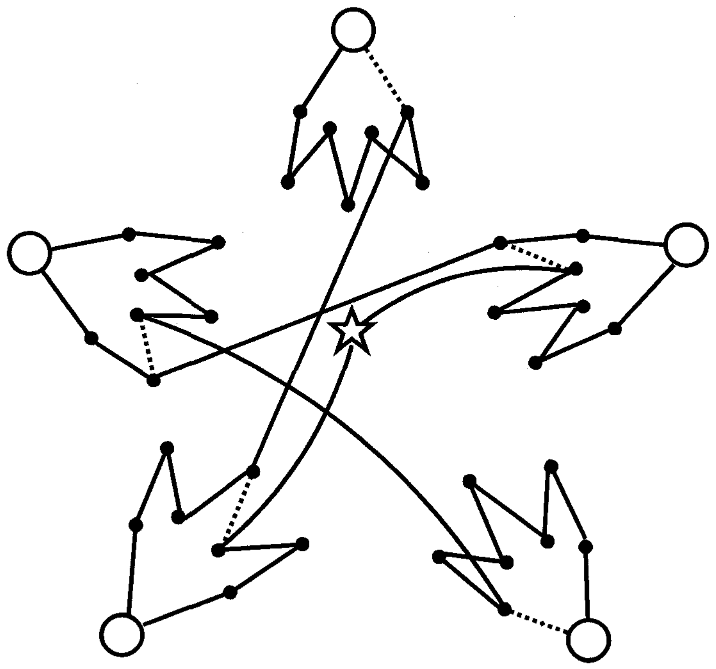


Figure 4.4:

This Hamiltonian path corresponds to a Hamiltonian Cycle in M_k, by Déjter's Theorem.

4.2 Another Possible Reduction

In the previous section we described a method that at least conceptually reduces part of the problem. The issue with that method is that we never specified how to construct Z_k or which vertices specifically were in it. Methods of choosing which vertices are in Z_k may be time consuming and inefficient. Depending on which method we choose it may actually be more time consuming to construct Z_k than to just simply use a Hamiltonian path heuristic to search through R_k. Luckily, we do not need to construct Z_k in order to utilize the symmetries of R_k to reduce the problem. If we have a cycle in R_k which includes exactly one vertex from each non-central compatibility class and includes no central vertices, then we can build k compatible cycles and interconnect them just like in the previous method.

Pósa [20] developed a useful Hamiltonian path heuristic which was used in [25],[24] to construct Hamiltonian paths in R_k. The heuristic starts off at some vertex, v_1. This vertex will be at the start of our path. Then the heuristic keeps extending the path by randomly choosing an available neighbor of the vertex at the end of the path. It does this until there are no available neighbors with which to extend the path. If v_m is the vertex at the end of the path, $v_1, v_2, ..., v_m$ and v_m has no available neighbors, then the neighbors of v_m must all be on the path already. Say v_m, is adjacent to v_i, with $1 < i < m-1$. We observe that $v_1, v_2, v_3, .., v_i, v_m, v_{m-1}, v_{m-2}, .., v_{i+1}$ is also a path of length m. However, it may be possible to extend this new path, unlike its predecessor, since the new path does not have v_m at its end. We can keep doing this until we get stuck and then backtrack and try to extend the path again. This heuristic was used successfully to find a Hamiltonian path in R_{17}, which is a graph with 129,644,790 vertices.

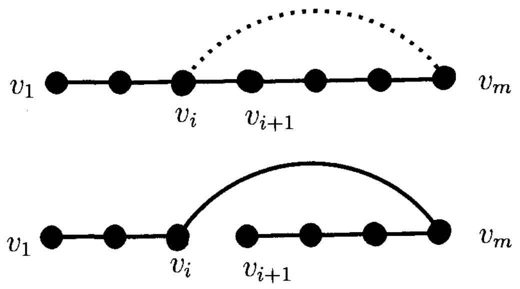


Figure 4.5:

We need to modify this heuristic slightly when we search through R_k since we want to utilize the symmetries of R_k. Working in a case were n and k are both prime we first remove the two central vertices from our search space. Then starting with an l-vertex we start extending a path from that vertex. Every time we add a vertex to the path, we remove that vertex as well as all vertices that are compatible with it from the search space. This guarantees that the path does not contain two vertices that are compatible with one another. Unlike in Pósa's algorithm, when we cannot extend the path, we are not guaranteed that the neighbors of the vertex at the end of the path are on the path. However, we are guaranteed that each neighbor is on a path which is equivalent under some CFA. So our current path, say $v_1, v_2, ..., v_m$, cannot be extended and the vertex at the end of the path, v_m, is adjacent to a vertex, w_i, with $1 < i < m-1$, which is on some path, $w_1, w_2, ..., w_m$, which is equivalent under some CFA to $v_1, v_2, ..., v_m$. Notice now that $w_1, w_2, w_3, .., w_i, v_m, v_{m-1}, v_{m-2}, .., v_{i+1}$ is also a path of length m and it might be possible to extend this path, unlike its predecessor. Also note that this path contains exactly one vertex from each compatibility class that we have already removed and still starts with an l-vertex. If successful, this algorithm is faster than Pósa's algorithm, because it needs to construct a much shorter path.

We modify the heuristic further to make sure the path is a cycle. This can be done by:

1) We can check if the point at the end of the path is adjacent to the l-vertex at the beginning of the path. If it is, we have a Hamiltonian cycle in Z_k, as desired. If this is not the case, we backtrack and try again. The heuristic applied in [25],[24] used this idea to make sure their path had the desired endpoints.
2) When the path contains all vertices we can still use Pósa's technique to change which vertex is at the end of the path, as in the diagram on page 56. We do this repeatedly until the path has the desired end vertex.

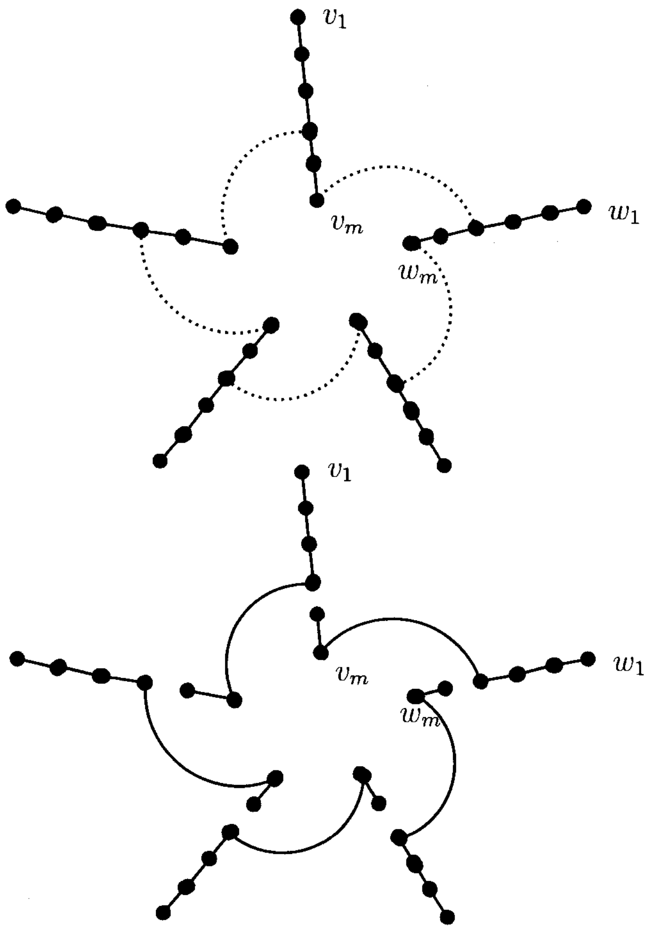


Figure 4.6:

Once we have constructed a Hamiltonian cycle in Z_k, we proceed as in the previous section to interconnect the k cycles we have created as well as the two central vertices. We do this with a different modification of the Pósa heuristic. We start with 11^k0^k and break the cycle that contains it into a path which starts with $11^k0^k = (k)$. We will refer to the vertex at the end of the path as E. This is our initial path. Now, if

E has a neighbor, A, in a cycle, C, that we have not yet added to our current path, then we can break C into a path by removing one of the two edges incident with A, creating a path C'. Then we attach A to E, there by extending our current path. If E has no such neighbor then all of its neighbors are on the current path and we use Pósa's method to choose a different path with a different vertex at its end. We do this until we have a path containing all vertices with the endpoints 11^k0^k and $1(10)^k$. Using this method allows us to add many vertices at a time to our current path. It also places fewer restrictions on the order of the vertices in the final path because this does not force them to appear in the same order that they did in the cycles we constructed earlier.

4.3 Implementation

We coded the algorithm described in section 4.2. The java code is included in Appendix A. This code successfully constructed a Hamiltonian path in R_5 (the smallest case it can be tried on) and R_{11}, verifying that the algorithm works. The code output for R_5 is included in Appendix B. The next case where this algorithm can be applied is R_{23} and this case is still open. Due to time and memory constraints we did not try the algorithm on this case. Also, we did not recode the algorithm used in [25] for direct comparison on the same machine. Hence, we do not have direct proof that our algorithm is faster. We believe that is it faster based on the fact that it operates on a much smaller graph than R_k.

4.4 Some Possible Improvements the Algorithm

It may be worth noting that in our runs of the algorithm, no backtracking was used at all. In spite of that, the algorithm still managed to find an appropriate Hamiltonian path each time it was implemented. It is likely that using strategic backtracking in this algorithm, both when constructing and interconnecting the cycles, would considerably improve runtimes. This would involve backtracking whenever several successive rearrangements do not yielded an extendible path. Our current method just keeps rearranging the path until it is possible to extend it. If we treat our representative vertices in R_k as binary integers, note that when generating the list of vertices in R_k as described in section 3.7, the list is built in order from the smallest to the largest integer. This means that if we are searching for a vertex in that list we can do it using a binary search instead of a linear one which is what was done in our code. In the algorithm used in [25], instead of rearranging the list of vertices which represent the path, every time the path could not be extended a list of pending rearrangements was updated. Using this list all the rearrangements were implemented simultaneously. This lead to a significant improvement in runtime. It seems feasible to do something similar to improve our algorithm. It is our hope that implementing all these improvements in our algorithm will allow it to successfully find a Hamiltonian path in R_{23}. We also, note that it seems feasible to run a similar algorithm on cases where n and k are not both prime. We will just have more central vertices to find and keep track of. Hence it may be possible that a modification of our algorithm could be used in establishing that M_{18} is Hamiltonian.

Chapter 5

Applications to Gray Codes

We will mention one particular application of Hamiltonian cycles, Gray codes. We start by describing an application based on Gray codes and then we introduce them.

Consider a rotating device split into sections with a sensor on it which sends a binary signal to a computer telling it which section the sensor is currently on and hence the current position of the device. You can think of this device as a drum in a photocopy machine, a magnetic storage device or an axle of a vehicle (abs system); all these are real life examples of devices that use this idea. First, we use binary numbers as in part (a) of the above diagram to number the positions of the device:

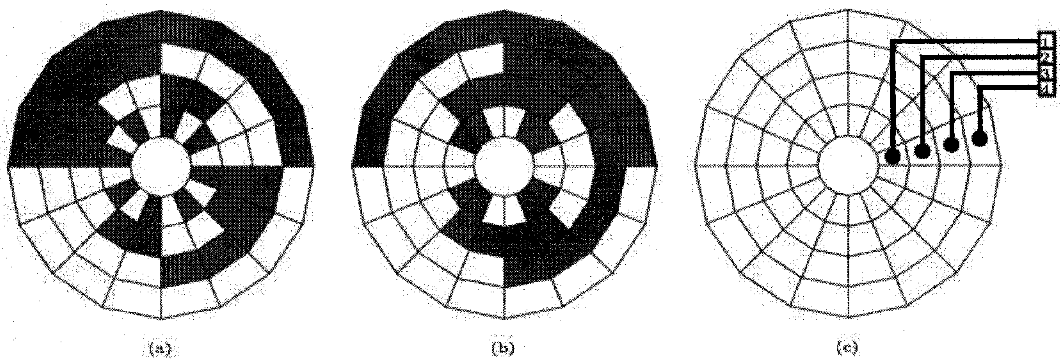


Figure 5.1:

position	binary representation	position	binary representation
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

Table 5.1:

If the device stops spinning and the sensor is in-between positions then the signal sent to the computer will be a mix of the two adjacent strings. In the above list of binary strings, we notice that certain adjacent positions differ significantly. For example, the 0^{th} position (0000) and the last position (1111) are adjacent, so that if the sensor is in-between those two positions, then it is possible that the signal sent to the computer could be about any one of the 16 positions. So it would be useful to have a circular list of binary strings such that any two strings which are adjacent on the list differ in exactly one position. If the device is labeled in this way, then when the sensor is between positions, the sensor sends a signal which is one of the positions which it is in-between, hence giving the computer a much more accurate information about the position of the device. The following is a list of binary strings with this minimal change property, (this one corresponds to part (b) of the diagram):

position	binary representation	position	binary representation
0	0000	8	1100
1	0001	9	1101
2	0011	10	1111
3	0010	11	1110
4	0110	12	1010
5	0111	13	1011
6	0101	14	1001
7	0100	15	1000

Table 5.2:

These minimal change listings are commonly referred to as "Gray codes" named after Frank Gray who was one of the first people to research them. It is easy to show that Gray codes exist for all possible lengths of binary string. We will include the proof of this well-known fact, because we will need it for the discussion that follows.

THEOREM 5.1.1 *[11] A listing of all binary strings of length n exists such that any two binary strings which are adjacent on the list differ in exactly one position (the first and last entries in the list are also considered adjacent).*

Proof: We will proceed by induction on n. For $n = 1$, the list $0, 1$ has the desired property. Now, assume inductively that listings with the desired adjacency properties exist for all $n < N - 1$. This means that there exists such a list, L_{N-1}, of all binary strings of length $N - 1$. Now we build L_N in the following way: add a zero to the beginning of every element in L_{N-1}, this creates a new list L'_{N-1}. Now take the list of the elements of L_{N-1} in reverse order and add a 1 to the beginning of each of those strings; call this list L''_{N-1} . Combine L''_{N-1} and L'_{N-1} into one list which we will call L_N. Since every N-bit binary string can be created by adding either a 0 or 1 to the beginning of one of the strings of L_{N-1}, L_N contains all possible strings of length N. Notice that all strings of L''_{N-1} have the adjacency property and so do the strings in L'_{N-1}. Since the first string in L'_{N-1} and the last string in L''_{N-1} differ in only the first bit and the first string in L''_{N-1} and the last string in L'_{N-1} differ in only the first bit, every pair of adjacent strings on the list, differ in exactly one position, as desired. □

The above proof shows us how to construct Gray codes of all different sizes recursively. Starting with 0,1 $\rightarrow$ 00,01,11,10 $\rightarrow$ 000,001,011,010,110,111,101,100 etc... Gray codes constructed in this way are called **binary reflected Gray codes**. Although these are useful, sometimes in practice it become useful to put additional restrictions on these lists. For example, note that, in a binary reflected Gray code, the first bit changes only twice which is less than any other bit whereas the last bit changes most often. There are situations in which that may be undesirable and we want a Gray code in which every bit changes the same number of times. Such balanced Gray codes were shown to exist for all even lengths of a binary string by Wagner and West [28]. For a general binary string some heuristics have been proposed. There are many variations on the theme of creating Gray codes with various different restrictions. Currently, the Middle Levels problem seems to be the most notorious unsolved Gray code problem.

Chapter 6

Conclusion

We developed two new characteristics of l-vertices (Lemma 3.4.10 and Theorem 3.4.4) as well as defined and characterized central vertices (Note 3.4.18). Then we used these characteristics to verify how many of these vertices are in particular R_k's (Theorem 3.5.9 and Observation 3.5.5), focusing on the case where both n and k are prime. We explored the role of these vertices in the automorphisms of R_k (Section 3.6) and used central vertices to define a particular type of automorphisms, CFAs. These CFAs, as well as the work of Pósa and Déjter played a crucial role in the development of the algorithm described in section 4.2. We believe that the algorithm is amenable to confirm more cases as Hamiltonian. It is also our hope that our work on the automorphisms of R_k will aid the development of a complete solution to the Middle Levels problem.

Bibliography

[1] **R. Cada, T. Kaiser, M. Rosenfeld and Z. Ryjacek**, Hamiltonian decompositions of prisms over cubic graphs, *Discrete Mathematics* 286(2004), 45-56.

[2] **Y. Chen**, Kneser graphs are Hamiltonian for $n \geq 3k$, *Journal of Combinatorial Theory, Series B* 80(2000), 69-79.

[3] **F. Chung, P. Diaconis, R. Graham**, Universal cycles for combinatorial structures, *Discrete Mathematics* 110(1992), 43-59.

[4] **I.J. Dejter**, Hamilton cycles and quotients of bipartite graphs, *Graph Theory with Applications to Algorithms and Computer Science* (1985), 189-199.

[5] **I.J. Dejter and J. Quintana**, Long cycles in revolving door graphs, *Congressus Numerantium* 60(1987), 163-168.

[6] **I.J. Dejter, J. Cordova and J. Quintana**, Two Hamilton cycles in bipartite reflective Kneser graphs, *Discrete Mathematics* 72(1988), 63-70.

[7] **I.J. Dejter, J. Cedeno and V. Jauregui**, A note on Frucht diagrams, Boolean graphs and Hamilton cycles, *Discrete Mathematics* 114(1994), 131-135.

[8] **G.A. Dirac**, Some theorems on abstract graphs, *Proceedings of the London Mathematical Society* 2(1952), 69-81.

[9] **D.A. Duffus, H.A. Kierstead and H.S. Snevily**, An explicit 1-factorization in the middle of the boolean lattice, *Journal of Combinatorial Theory, Series A* 65(1994), 334-342.

[10] **L. Euler**, Solutio d'une question curieuse qui ne paroit soumise á aucune analyse, *Mem. Acad. Sci. Berlin* 15(1759), 310-337.

[11] **F. Gray**, Pulse code communications, *U.S. Patent Number 2632058* (March 1953).

[12] **I. Havel**, Semipaths in directed cubes, *Graphs and Other Combinatorial Topics* (1983), 101-108.

[13] **P. Horak, T. Kaiser, M. Rosenfeld and Z. Ryjacek**, The prism over the middle-levels graph is Hamiltonian, *Order* 22 (2005) 73-81.

[14] **B. Jackson**, Hamiltonian cycles in regular 2-connected graphs, *Journal of Combinatorial Theory, Series B* 29 (1980), 27-46.

[15] **H.A. Kierstead and W.T. Trotter**, Explicit matchings in the middle levels of the boolean lattice, *Order* 5 (1988), 163-171.

[16] **J. Moon and L. Moser**, On Hamiltonian cycles in bipartite graphs, *Israel Journal of Mathematics* 1(1963), 163-165.

[17] **L. Lovász**, Problem 11. In *Combinatorial Structures and their Applications,* (1970).

[18] **O. Ore**, A note on Hamiltonian circuits, *The American Mathematical Monthly* 67(1960), 55.

[19] **O. Ore**, Hamiltonian connected graphs, *Journal de Mathématiques Pures et Appliquées* 42(1963), 21-27.

[20] **L. Pósa**, Hamiltonian circuits in random graphs, *Discrete Math.* 14(4) (1976), 359-364.

[21] **F. Ruskey**, Combinatorial Generation (2001). `www.1stworks.com/ref/RuskeyCombGen.pdf`

[22] **C.D. Savage**, Long cycles in the middle two levels of the Boolean lattice,*Ars Combinatoria* 35(A) (1993), 97-108.

[23] **C.D. Savage and P. Winkler**, Monotone Gray codes and the middle levels problem, *Journal of Combinatorial Theory, Series A*, 70(1995), 230-248.

[24] **C.D. Savage and I. Shields**, A Hamilton path heuristic with applications to the middle two levels problem, *Congressus Numerantium* 140(1999), 161-178.

[25] **C.D. Savage, I. Shields and B. J. Shields**, An update on the Middle Levels problem, *submitled* (Aug 2006).

[26] **I. Shields**, Hamilton cycle heuristics in hard graphs, *Ph.D. thesis*, North Carolina State University, Raleigh, (2004).
`www.lib.ncsu.edu/theses/available/etd-03142004-013420/`

[27] **R. Stanley**, Enumerative combinatorics vol. 1, (1997).

[28] **D.G. Wagner and J. West**, Construction of uniform Gray codes, *Congressus Numerantium.* 80(1991), 217-223.

Appendix A

Java code

```
package midlevel;

import java.util.ArrayList;
import java.util.Scanner;
import java.io.IOException;
import java.util.Random;

public class Main {
        private ArrayList<Node> node;

        /* this is the same k as in the thesis and is assigned a
         value in public MidLevel*/
        final private int K;

        // MAX is equal to k+1 and is assigned a value in public MidLevel
        final private int MAX;

        /* this is is the same n as in the thesis ( n=2k+1 )
         and is assigned a value in public MidLevel */
        final private int N;

        /* This is a list of all the elements in the
         subgroup of Z_n\{0} which has order k */
        final private int[] multiplyList;

        class Node extends Object {

                private ArrayList<Boolean> cell;

                @Override
                public boolean equals(Object object) {
                        Node node = (Node) object;
                        final int SIZE = this.cell.size();

                        for (int i = 0; i < SIZE; i++)
                                if (this.cell.get(i) != node.cell.get(i))
                                        return false;

                        return true;
                }
```

```
/* Default Constructor initiate a new ArrayList */
Node() {
        this.cell = new ArrayList<Boolean>();
}

/* Clone Constructor */
Node(Node clone) {
        this.cell = new ArrayList<Boolean>();
        for (int i = 0; i < clone.length(); i++)
                this.cell.add(clone.get(i));
}

/* @param value to be added to our Node String */
void add(boolean value) {
        this.cell.add(value);
}

boolean get(int index) {
        return this.cell.get(index);
}

/* Shift Node one placing */
void shift() {
        final int size = this.cell.size() - 1;
        final boolean temp = this.cell.get(size);

        for (int i = size; i > 0; i--)
                this.cell.set(i, this.cell.get(i - 1));
        this.cell.set(0, temp);
}

/* @param x is the multiplier */
void multiply(int x) throws IOException {
        if (x == 0)
                throw new IOException();
        final int size = this.cell.size();
        Node clone = new Node(this);
        int location;

        for (int i = 0; i < size; i++) {
                location = i * x % size;
                this.cell.set(location, clone.cell.get(i));
        }
        this.validify();
}

/* Flip the ones to zeros and zeros to ones */
void compliment() {
        final int SIZE = this.cell.size();
        for (int i = 0; i < SIZE; i++)
                if (this.cell.get(i) == true)
                        this.cell.set(i, false);
                else
                        this.cell.set(i, true);
}

/* Cheks if the number of ones exceeds the number of zeros
 in every prefix */
boolean valid() {
        int count = 0;
        final int SIZE = this.cell.size();
        for (int i = 0; i < SIZE; i++) {
                if (this.cell.get(i))
                        count++;
                else
                        count--;
                if (count <= 0)
                        return false;
        }
        return true;
}

/* Converts to vertex to a vertex in
 the same class which is valid */
void validify() {
        boolean flag = true;
        do
                if (!this.valid())
```

```
                                this.shift();
                        else
                                flag = false;
                while (flag);
        }

        ArrayList<Node> neighbours() throws IOException {
                Node clone;
                ArrayList<Node> list = new ArrayList<Node>();
                final int SIZE = this.cell.size();

                for (int i = 0; i < SIZE; i++) {
                        if (this.cell.get(i)) {
                                clone = new Node(this);
                                clone.cell.set(i, false);
                                clone.compliment();
                                clone.validify();
                                list.add(clone);
                        }
                }

                if (list.size() != MAX)
                        throw new IOException();
                return list;
        }

        ArrayList<Node> friends() throws IOException {
                Node clone;
                ArrayList<Node> list = new ArrayList();

                for (int i : multiplyList) {
                        clone = new Node(this);
                        clone.multiply(i);
                        clone.validify();
                        list.add(clone);
                }

                return list;
        }

        /* @return value number of ones in Node String */
        int numberOfOnes() {
                int count = 0;
                for (int i = 0; i < this.cell.size(); i++)
                        if (this.cell.get(i))
                                count++;

                return count;
        }

        /* @return value number of zeros in Node String */
        int numberOfZeros() {
                int count = 0;
                for (int i = 0; i < this.cell.size(); i++)
                        if (!this.cell.get(i))
                                count++;

                return count;
        }

        @Override
        public String toString() {
                String temp = "";
                for (int i = 0; i < cell.size(); i++)
                        if (cell.get(i) == true)
                                temp += "1";
                        else
                                temp += "0";
                return temp;
        }

        int length() {
                return this.cell.size();
        }
}

public Main(int value) throws IOException {
        //this initializes the variables used to what the user indicated
        K = value;
        MAX = K + 1;
        N = 2 * K + 1;
        node = new ArrayList<Node>();
        multiplyList = new int[K];
        multiplyList[0] = 4;
        for (int i = 1; i < K; i++)
                multiplyList[i] = 4 * multiplyList[i - 1] % N;

        if (multiplyList[multiplyList.length - 1] != 1)
                throw new IOException();
}

private void nodeBuilder(Node temp) throws IOException {
        /*this method is used to extend the partial strings when
         building the list of vertices in R_k*/
        if (temp.numberOfOnes() == this.MAX) {
                this.node.add(zeroSpaces(temp));
                // System.out.println(this.node.get(this.node.size() - 1));
        } else if (temp.numberOfOnes() > this.MAX)
                throw new IOException();
        else {
                if (temp.numberOfOnes() - 1 == temp.numberOfZeros()) {
                        temp.add(true);
                        nodeBuilder(temp);
                } else {
                        Node clone = new Node(temp);

                        temp.add(false);
                        nodeBuilder(temp);

                        clone.add(true);
                        nodeBuilder(clone);
                }
        }
}

private int find(ArrayList<Node> x, Node y) throws IOException {
    /*this method finds and returns the possition of the given node
     on the given list*/
        if (x.size() == 0)
                throw new IOException();
        for (int i = 0; i < x.size(); i++) {
                if (x.get(i).equals(y))
                        return i;
        }
        throw new IOException();
}

private boolean finder(ArrayList<Node> x, Node y) throws IOException {
    /*This method returns true if the given node is on the given list
     and false otherwise*/
        if (x.size() == 0)
                throw new IOException();
        for (int i = 0; i < x.size(); i++) {
                if (x.get(i).equals(y))
                        return true;
        }
        return false;
}

private ArrayList<Node> appender(ArrayList<Node> x, ArrayList<Node> y ){
    /*This combines the two lists given into one list*/
        ArrayList<Node> temp = x;
        for( int i = 0; i < y.size(); i++) temp.add(y.get(i));

        return temp;
}

private ArrayList<Node> cycleCutter(ArrayList<Node> x, int y, boolean
direction ){
    /*This method returns the nodes in a cycle starting as the given
     index and in the give direction*/

        ArrayList<Node> temp = new ArrayList<Node>();
        if( !direction ){

                for(int i=0; i<x.size();i++)
```

```
                        temp.add(x.get((y+i)%x.size()));

                }else{

                        for(int i=0; i<x.size();i++)
                                temp.add(x.get((y-i+x.size())%x.size()));
                }
        return temp;
}

private ArrayList<Node> flip(ArrayList<Node> list, int x, int y)
        /*This implements the modificed Posa flip to make a cycle */
throws IOException {
        if(y<2)return list;
        ArrayList<Node> temp = new ArrayList<Node>();
        final int multiplier = this.multiplyList[x];
        for (int i = 0; i < y + 1; i++)
                temp.add(list.get(i));

        for (int i = list.size() - 1; i > y; i--) {
                list.get(i).multiply(multiplier);
                temp.add(list.get(i));
        }
        return temp;
}

private ArrayList<Node> flip(ArrayList<Node> list, int y)
throws IOException {
    /*This method implements the other modified Posa flip which
     is used to change the end point of the current path then trying to
     interconnect the cycles */

        if(y<2)return list;
        ArrayList<Node> temp = new ArrayList<Node>();
        for (int i = 0; i < y + 1; i++)
                temp.add(list.get(i));

        for (int i = list.size() - 1; i > y; i--) {
                temp.add(list.get(i));
        }
        return temp;
}

private ArrayList<Node> purge(ArrayList<Node> x, ArrayList<Node> y)
throws IOException {
    /*This method suplise the first list given with all the common
     entries from the second list removed*/

        ArrayList<Node> result = new ArrayList<Node>();
        boolean inList = false;

        for (int i = 0; i < x.size(); i++) {
                for (int j = 0; j < y.size(); j++) {
                        if (x.get(i).equals(y.get(j))) {
                                inList = true;
                                break;
                        }
                }
                if (!inList)
                        result.add(x.get(i));
                inList = false;
        }

        return result;
}

private ArrayList<Node> purge(ArrayList<Node> x, Node y) throws
IOException {
        final int size = x.size();
        for (int i = 0; i < size; i++)
                if (x.get(i).equals(y)) {
                        x.remove(i);
                        break;
                }
        return x;
}

private ArrayList<Node> intersection(ArrayList<Node> x, ArrayList<Node> y)
```

```
throws IOException {
        ArrayList<Node> result = new ArrayList<Node>();
        boolean inList = false;

        for (int i = 0; i < x.size(); i++) {
                for (int j = 0; j < y.size(); j++) {
                        if (x.get(i).equals(y.get(j))) {
                                inList = true;
                                break;
                        }
                }
                if (inList)
                        result.add(x.get(i));
                inList = false;
        }

        return result;
}

private ArrayList<Node> multiList(ArrayList<Node> x, int y)throws
IOException{

        ArrayList<Node> temp= new ArrayList<Node>();
        Node node= new Node();

        for(int i=0; i< x.size(); i++){
                node = new Node( x.get(i) );
                node.multiply(y);
                temp.add(node);
        }
        return temp;
}

private Node zeroSpaces(Node node) {
    /* this is used in nodeBuilder to append the appropriate number
     of 0's to a string when it already had k+1 1's*/
        int adjust = this.N - node.length();
        if (adjust == 0)
                return node;
        for (int i = 0; i < adjust; i++)
                node.add(false);

        return node;
}

private static void print(ArrayList<Node> x) {
   // prints a list of vertices
        System.out.println( "Length = " + x.size());
        for (int i = 0; i < x.size(); i++) {
                System.out.println( x.get(i) );
        }
}

/* Entry point for our program. */
public static void main(String[] ignored) throws IOException {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter value of K: ");
        Main simulation = new Main(in.nextInt());
        simulation.run();
}

private void run() throws IOException {
        // Make the list of vertices in R_k
        Node first = new Node();
        first.add(true);
        first.add(true);
        this.nodeBuilder(first);
        // Make the two central vertices, then remove them from the list of
        // vertices R_k
        Node cent1 = new Node();

        for (int i = 0; i < N; i++) {
                cent1.add(false);
        }
        System.out.println("R_k made successfully");
        System.out.println("running Posa path constructor....");

        cent1.cell.set(0, true);
```

```
for (int i : multiplyList)
        cent1.cell.set(i, true);
Node cent2 = new Node(cent1);
cent2.cell.set(0, false);
cent2.compliment();
cent1.validify();
cent2.validify();

ArrayList<Node> cent = new ArrayList<Node>();
cent.add(cent1);
cent.add(cent2);

/* compare is a copy of the list of all the vertices in
 R_k 'node' which we will use to check that all the vertices
 are on our final Hamiltonian path*/

ArrayList<Node> compare = new ArrayList<Node>(node);

node = purge(node, cent);
// Now we the list that will turn into the cycle in Z_n
ArrayList<Node> cycle = new ArrayList<Node>();
cycle.add(node.get(node.size() - 2));
cycle.add(node.get(node.size() - 1));

node = purge(node, cycle.get(0).friends());
node = purge(node, cycle.get(1).friends());

// Initialize variables used in main loop;
ArrayList<Node> inter = new ArrayList<Node>();
ArrayList<Node> flipList = new ArrayList<Node>();
int location = -1;
int multiIndex = -1;
Node choosen = new Node();
Node flipNode = new Node();
boolean flip = false;
boolean allGood = true;
boolean done = false;
boolean pathNowCycle = false;
Random rand = new Random();
while (!done) {
        while (!flip) {
                inter = intersection(node, cycle.get(cycle.size() - 1)
                                .neighbours());
                if (inter.size() != 0) {
                        choosen = inter.get(rand.nextInt(inter.size()));
                        cycle.add(choosen);
                        node = purge(node, choosen.friends());

                } else {
                        if (node.size() == 0) {
                                done = true;
                                System.out.println("done");
                                break;
                        } else {
                                flip = true;
                        }
                }
        }
        if (!done) {
                flipList = purge(cycle.get(cycle.size() - 1).neighbours(),
                                cycle.get(cycle.size() - 2));
                flipNode = flipList.get(rand.nextInt(flipList.size()));
                flipList = intersection(cycle, flipNode.friends());
                if (flipList.size() == 0){}
                        //Do nothing
                else{
                        multiIndex = find(flipNode.friends(), flipList.get(0));
                        flipNode = flipList.get(0);
                        location = find(cycle, flipNode);
                        cycle = flip(cycle, multiIndex, location);
                        flip = false;
                }
        } else {
                /* verify that cycle was built correcrly*/
                for (int i = 0; i < cycle.size() - 1; i++) {
                        if (!finder(cycle.get(i).neighbours(), cycle.get(i + 1))) {
                                System.out.println("error in cycle construction");
                                allGood = false;
                        }
                }
                if (allGood) {
                        System.out.println("Path contains all vertices:");
                        System.out.println("Attempting to make path into cycle...");

                        do {

                                if (!finder(cycle.get(cycle.size() - 1).neighbours(),
                                                cycle.get(0))) {
                                        //System.out.println("path... to ... cycle");
                                        flipList = purge(cycle.get(cycle.size() - 1)
                                                        .neighbours(), cycle.get(cycle.size() - 2));
                                        flipNode = flipList.get(rand.nextInt(flipList
                                                        .size()));
                                        flipList = intersection(cycle, flipNode.friends());
                                        if (flipList.size() == 0) {
                                                System.out.println("flipList error");
                                        } else {
                                                multiIndex = find(flipNode.friends(), flipList
                                                                .get(0));
                                                flipNode = flipList.get(0);
                                                location = find(cycle, flipNode);
                                                cycle = flip(cycle, multiIndex, location);
                                        }
                                } else {
                                        pathNowCycle = true;
                                        allGood = true;
                                        for (int i = 0; i < cycle.size() - 1; i++) {
                                                if (!finder(cycle.get(i).neighbours(), cycle
                                                                .get(i + 1))) {
                                                        System.out
                                                        .println("error in cycle construction");
                                                        allGood = false;
                                                }
                                        }
                                }
                        } while (!pathNowCycle);
                        if (allGood) {
                                System.out.println("cycle complete");
                        }
                }
        }

}

System.out.println("Interconnecting borken cycles....");
ArrayList<Node> finalPath = new ArrayList<Node>();//Hamiltonian path
/*This is a array that contains all the cycles that are equivalent
 Under CFA's as well the last entry on the array is a list of the
 two central vertices */

ArrayList<Node>[] CYCLES = new ArrayList[K + 1];
CYCLES[0]=cycle;
boolean direction;
int numberOfCyclesIncluded;
for(int i=0; i<K;i++){
        CYCLES[i+1] = multiList(cycle,this.multiplyList[i]);
}
CYCLES[K]=cent;
ArrayList<Node> NeighborList = new ArrayList<Node>();

direction = rand.nextBoolean();
finalPath = appender( finalPath,cycleCutter(CYCLES[0], 1,direction ) );
numberOfCyclesIncluded=1;
done=false;
Node neighbor =new Node();
int index;
do{
    NeighborList = purge(finalPath.get(finalPath.size()-1).neighbours(), finalPath);

    if(numberOfCyclesIncluded == K+1) done=true;
    else if(NeighborList.size()==0 ){
        /*if this is the case all neighbors of the last vertex on
         our current path  lie on the path so we Posa flip the path*/
        NeighborList = finalPath.get(finalPath.size()-1).neighbours();
        neighbor = NeighborList.get(rand.nextInt(NeighborList.size()));
        finalPath = flip(finalPath, find(finalPath,neighbor));
```

```
                }else{
                    /*If this is the case the last vertex on our current path
                     has a neighbor on a path we haven't added yet so we can
                     extend the current path by*/
                    //find on cycle list;
                    neighbor  = NeighborList.get(rand.nextInt(NeighborList.size()));
                    int j=0;
                    do{
                        if( finder(CYCLES[j],neighbor) ) {
                            index =  find(CYCLES[j],neighbor);
                            direction = rand.nextBoolean();
                            finalPath = appender( finalPath,
                                    cycleCutter(CYCLES[j],index,direction ) );
                            numberOfCyclesIncluded++;
                            break;
                        }
                        j++;
                    }while(true);
                }
            }while(!done);
            System.out.println("Path now contains all verties");
            System.out.println("flipping path to get correct endpoints...");
            boolean correctEndpoints = false;
            Node end =new Node();
            end.add(true);
            for(int i = 0; i < K; i++){
                end.add(true);
                end.add(false);
            }

            do{
                if(finalPath.get(finalPath.size()-1).equals(end)) correctEndpoints =true;
                else{
                    NeighborList = finalPath.get(finalPath.size()-1).neighbours();
                    neighbor = NeighborList.get(rand.nextInt(NeighborList.size()));
                    finalPath = flip(finalPath, find(finalPath,neighbor));

                }
            }while(!correctEndpoints);

            System.out.println("Done making Hamiltonian Cycle in R_k");
            /*Check that the final path is infact Hamiltonian by
             1) checking that all the vertices are in it
             2) all adjacent enties are neighbors*/

            if( purge(compare,finalPath).size() == 0 &&
                    compare.size() == finalPath.size() )
            {
                System.out.println("Final path contains all vertices");
                allGood = true;
                for (int i = 0; i < finalPath.size() - 1; i++)
                {
                    if (!finder(finalPath.get(i).neighbours(),
                            finalPath.get(i + 1)))
                    {
                        System.out.println("error in cycle construction");
                        allGood = false;
                    }

                }
                if( allGood ) print(finalPath);
            }
        }
    }
```

Appendix B

Code output

R_5:

```
Enter value of K:
5
R_k made successfully
running Posa path constructor....
done
Path contains all vertices:
Attempting to make path into cycle...
cycle complete
Interconnecting borken cycles....
Path now contains all verties
flipping path to get correct endpoints...
Done making Hamiltonian Cycle in R_k
Final path contains all vertices
Length = 42
11111100000
11111010000
11011110000
11110010010
11011010010
11010110100
11101010010
11110001010
11101010100
11101100010
11101100100
11011001100
11011100100
11111001000
11101110000
11110011000
11100111000
11110001100
11100110010
11010101100
11011001010
11011010100
11010110010
11101001010
11110101000
11010111000
11100101100
11101001100
11011100010
11101101000
11111000100
11110110000
11110010100
11101011000
11100110100
11011011000
11110100100
11111000010
11110100010
11011101000
11100101010
11010101010
```

R_{11}

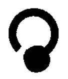
Appendix C

Diagrams of M_k's and R_k's

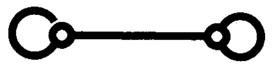
C.1 R_k

All l-vertices are pictured with loops. Central vertices are pictured as empty circles. The remaining vertices are pictured as filled circles.

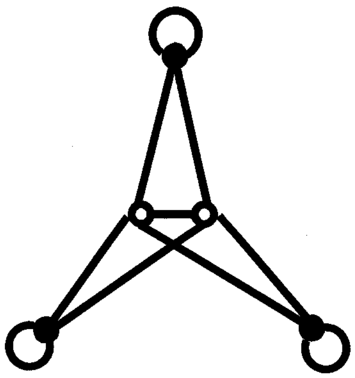
R_1 :



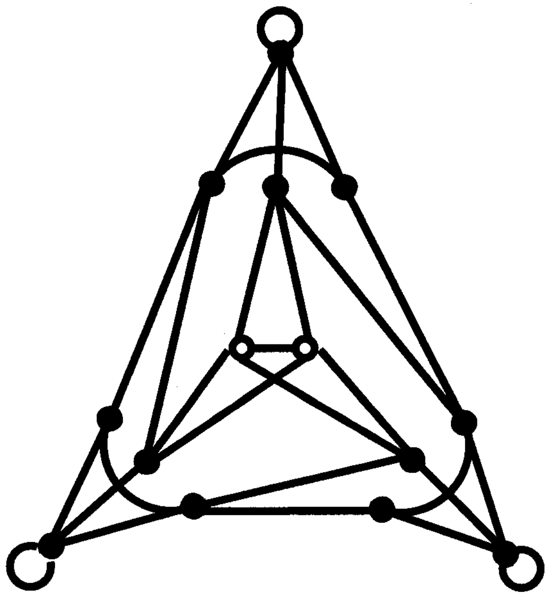
R_2 : Note that this is the only case were there are vertices that are both central and l-vertices.



R_3 :



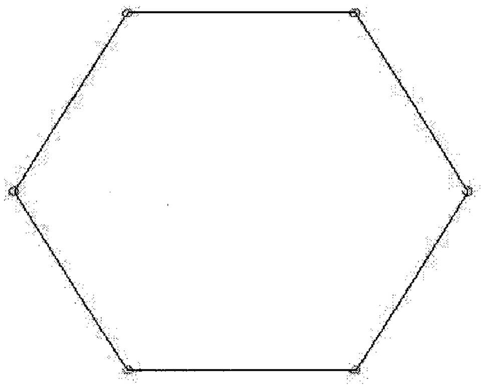
R_4 :



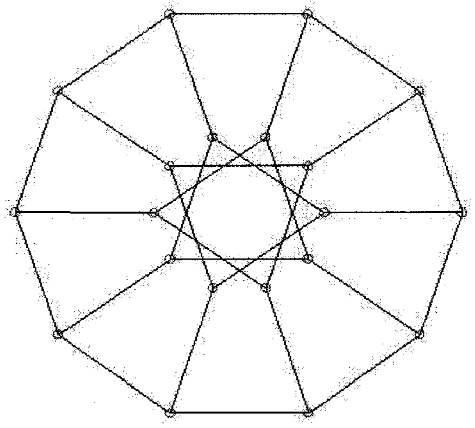
C.2 M_k

Because M_k is vertex transitive we choose not to label l-vertices or central vertices any differently than other vertices.

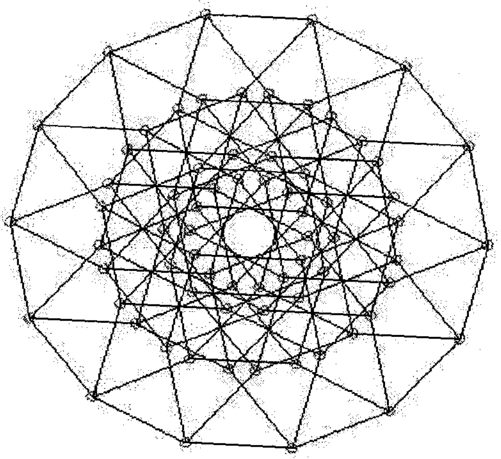
M_1 :



M_2 :



M_3 :



M_4 :