rijksuniversiteit groningen

faculteit Wiskunde en Natuurwetenschappen

Zero Forcing Sets for Distance Regular Graphs

Bachelor thesis Mathematics

June 2014

Student: R.A. Oosterman

First supervisor: Dr. M.K. Camlibel

Second supervisor: Prof. Dr. H. Waalkens

Abstract

In this thesis, linear systems on graphs are defined, representing a multi-agent system with leader and follower agents. The purpose is to render such a system controllable, and to reach this, one has to assign vertices as leader agents in a specific way. To do so, a number of algorithms are described in this thesis, and tested on several known examples of graphs known as distance regular graphs. Comparing to some properties of these graphs, one can finally decide whether the found leader set is optimal or not.

Contents

Chapter 1

Introduction

In many fields of science, networks consisting of several operating components can be found. Their members operate individually, but can interchange information with each other. Such members are often named as *agents*, and they are connected to each other in a certain way, depending on the purpose of the network. The agents all have a certain state, which can be changed due to information they obtain from other agents. Depending on the application of these networks, the agents can adapt their state to the surrounding agents, in order to reach the purpose of the network.

For instance, one can think of a collection of drones, flying in a certain formation, and responding to each other when the velocity or direction of one of its components is changed instantly. Also, one can see accounts on social media, such as Facebook, as agents that are connected in a certain manner (the " friends " of the accounts coincide with the connections between the agents). These accounts interchange information with each other, for instance to respond to the behaviour of surrounding accounts. More information about these and other applications can be found in [2, Section 1.2].

Systems in which such a networked structure of leader and follower agents is involved, are known as *multi-agent systems*. Such systems can be represented mathematically by a *graph*, that defines how many agents the network contains (the *vertices* or *nodes* of the graph) and how these are connected (the *edges* of the graph).

In order to make the state of the system externally controllable, several agents need information from outside the system. These agents are known as *leader agents* or *leaders*, and their role is to use this information to communicate with the other agents. Only these leader agents obtain information from outside the system (mathematically we often speak of input), and they are able to change their state due to this input. The agents that only gain information from other agents, and do not obtain input from outside the network, are said to be *follower agents*.

It seems obvious that assigning each agent as a leader agent will render the system externally controllable. However, giving input to a multi-agent system

takes energy, which is limited in most of the applications of such systems [2, section 1.1]. Therefore, it is desirable that the number of chosen leader agents is as few as possible.

Consequently, imagine a certain network of agents, represented by given graph. We want the system to be controllable, but the number of leader agents to be as few as possible. To maintain this, in which way do we have to assign vertices as leader agents? This question will form the main problem of this thesis.

In order to obtain an answer to this seemingly simple problem, we need some more mathematical foundation about graph theory and mathematical systems theory. Their preliminaries will be provided in section 2.1 respectively section 2.2. With this knowledge in mind, we can define linear systems on graphs (section 3.1). From that, we can give a mathematical interpretation of our problem, namely the problem of finding *zero forcing sets* (section 3.2). It turns out that the system is controllable if and only if the chosen leaders form a zero forcing set.

It will become clear that this problem is not as simple as it appears to be, since it is not (yet) possible to find the minimal number of leader agents in a straightforward way. In fact, it turns out that it is not even possible to check for a certain collection of leader agents whether it is optimal or not!

It is possible, however, to verify whether a chosen collection of leaders is indeed minimal. We will do this by using several properties of a graph, such as the *minimal rank* of a graph, or the *maximal eigenvalue multiplicity*. This approach is explained in section 3.3.

One class of graphs that will be treated particularly in this thesis, is the class of *distance regular graphs*. These graphs contain a specific structure when speaking of the distance between two nodes (that is, how many connections are needed to reach a certain node from another one). The terminology behind this kind of graphs is mentioned in section 3.4.

Since it is not possible to find an optimal collection of leaders in a straightforward manner, we have to try to solve our problem by using heuristic algorithms. In chapter 4, several of such algorithms are introduced, applying to distance regular graphs in general. These algorithms will be used on several graphs of which they are known to be distance regular, in order to find a reasonable set of leaders. It turns out that the result depends on the chosen algorithm, but none of the algorithms always produces the least number of leaders. For some graphs however, it can be shown that their found number of leaders is minimal, and therefore the problem can be solved for some graphs indeed!

Chapter 2

Preliminaries

2.1 Graph theory

In this thesis, we are interested in undirected graphs. Such a *graph* G is given by $G = (V, E)$ where $V = \{1, 2, \ldots, n\}$ is the *vertex set* and $E \subseteq V \times V$ is the *edge set*. The number of vertices of a graph G is called the *cardinality* of the graph, and is denoted by $|G|$.

We assume that there are no self-loops, that is, for every $v \in V$, $(v, v) \notin E$. Furthermore, we consider undirected graphs, that is, $(v, w) \in E \iff (w, v) \in E$. Two vertices v and w for which $(v, w) \in E$ are said to be *adjacent* to each other or *neighbours*. The *degree* of a vertex v is the number of vertices that are adjacent to this vertex, and is denoted by $d(v)$.

We often use the following matrices associated to a graph $G = (V, E)$. The *adjacency matrix* A is defined as follows:

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{if } (i, j) \notin E. \end{cases} \tag{2.1}$$

The *degree matrix* D is a diagonal matrix with degrees of vertices being diagonal elements, that is

$$D(G) = \begin{pmatrix} d(1) & 0 & \cdots & 0 \\ 0 & d(2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d(n) \end{pmatrix} \tag{2.2}$$

The *Laplacian* L is given by $L = D - A$. To avoid possible ambiguity, we sometimes write $A(G)$, $D(G)$ and $L(G)$ to make the dependency of G more clear.

A *path* between two vertices v and w is a sequence of vertices, say $v_0, v_1, \ldots, v_k$, with $v_0 = v$, $v_k = w$ and $(v_i, v_{i+1}) \in E$ for each i such that $0 \leq i \leq k - 1$. Such a path is said to have *length* k. A graph G is said to be *connected* if there exists a path between every pair of vertices $v, w \in V$.

The *distance* between two vertices is the length of the shortest path between them. The distance between the vertices v and w is denoted by $\text{dist}(v, w)$. By definition we say $\text{dist}(v, v) = 0$ for each $v \in V$. The *diameter* of the graph G, denoted by $\text{diam}(G)$, is the largest possible distance between two vertices of G, that is,

$$\text{diam}(G) = \max_{v,w \in V} \text{dist}(v, w). \tag{2.3}$$

A *partition* of a graph G is a collection of sets of vertices, which are called *cells*. The partition itself is denoted by $\pi = \{C_0, \ldots, C_{m-1}\}$, where $C_k \subseteq V$ denotes a cell of the partition. These cells are constructed in such a way that each vertex is only contained in one cell, that is,

$$C_i \cap C_j = \emptyset, i \neq j \tag{2.4}$$

$$\bigcup_k C_k = V. \tag{2.5}$$

The *cardinality* of a partition π is the number of cells C_k the partition contains, and is denoted as $\text{card}(\pi)$. This notion should not be confused with the cardinality of graphs, which denotes the number of vertices inside a graph. One kind of partition deserves special attention.

Definition 2.1.1. [3, page 6] The *distance partition* relative to a vertex v corresponding to a graph $G = (V, E)$, denoted by $\pi_D(v)$, is the partition $\{C_0, C_1 \ldots, C_{\text{diam}(G)}\}$, where

$$C_k = \{u \in V \,|\, \text{dist}(u, v) = k\} \text{ for } 0 \leq k \leq \text{diam}(G). \tag{2.6}$$

2.2 Systems theory

Consider the input/state linear system

$$\dot{x} = Ax + Bu, \tag{2.7}$$

where $x \in \mathbb{R}^p$ is called the state and $u \in \mathbb{R}^q$ the input. Furthermore, A and B are matrices of appropriate sizes. The *solution* of (2.7) for the initial condition $x(0) = x_0$ and input u is denoted by $x(t, x_0, u)$.

Definition 2.2.1. [1, Definition 4.13] A linear system (2.7) is said to be *controllable* if for any two states $x_0, x_1 \in \mathbb{R}^p$ there exist a finite time $t_1 \geq 0$ and an admissible input function u such that $x(t_1, x_0, u) = x_1$.

Controllability of a linear system can be characterised as the following classical theorem states.

Theorem 2.2.1. [1, page 62-71] The following statements are equivalent:

1. The system (2.7) is controllable.
2. The matrix $\begin{bmatrix} B & AB & \cdots & A^{p-1}B \end{bmatrix}$ has full row rank.
3. The matrix $\begin{bmatrix} A - \lambda I & B \end{bmatrix}$ has full row rank for each eigenvalue λ of A.

The last condition is known as the *Hautus test* for controllability.

Chapter 3

Systems defined on graphs

3.1 Qualitative classes of graphs

In this thesis, the controllability of linear systems that are defined on graphs is studied. In order to clarify what we mean by systems defined on graphs, we need to introduce some terminology.

Definition 3.1.1. The qualitative class $\mathcal{Q}(G)$ of a graph G is given by the following set of matrices:

$$\mathcal{Q}(G) = \{X \in \mathbb{R}^{|G|\times|G|} | X_{ij} \neq 0 \iff (i,j) \in E(G)\}. \tag{3.1}$$

Note that the diagonal elements of members of the qualitative class are not restricted. Also note that, according to section 2.1 the adjacency matrix and the Laplacian matrix of a graph G actually belong to its qualitative class.

In chapter 1, there has been discussion about leader and follower agents. Given a graph G, the set of vertices corresponding to leader agents is said to be the *leader set* V_L, and is given by

$$V_L = \{\ell_1, \ldots, \ell_m\}, \tag{3.2}$$

where $m \leq n$ and $l_i \in V$ for each i. The remaining vertices should correspond to the follower agents, and together form the *follower set* V_F, given by

$$V_F = V \setminus V_L. \tag{3.3}$$

By systems defined on a graph, we mean the linear systems of the form

$$\dot{x} = Xx + Uu \tag{3.4}$$

where $x \in \mathbb{R}^n$ is the state with the same dimension as the cardinality of G, and $u \in \mathbb{R}^m$ is the input with the dimension equal to the number of elements of V_L. The matrices X and U are assumed to have appropriate sizes. The only demands are that $X \in \mathcal{Q}(G)$ and U should correspond to the leader set V_L in the following way:

$$U_{ij} = \begin{cases} 1 & \text{if } i = \ell_j \\ 0 & \text{if } i \neq \ell_j. \end{cases} \tag{3.5}$$

As mentioned in section 3.1, the Laplacian matrix L of a graph G is a suitable substitute for the matrix X. For instance, this problem can be extended to multi-agent systems with single integrator dynamics, such as $\dot{x} = -Lx + Mu$, where L is the Laplacian and $M = U$, or even simpler systems such as

$$\dot{x}_i = \begin{cases} x_i & \text{if } i \notin V_L \\ x_i + u_i & \text{if } i \in V_L \end{cases} \tag{3.6}$$

Note that each x_i is a scalar variable. One can also create multi-agent systems in which not all vertices are contained in the same linear system, but each separate vertex is associated to a linear system on itself. In [3, section II.A], such systems are described to be of the form

$$\dot{x}_i = Ax_i + Cz_i \tag{3.7}$$

if a vertex i is a follower agent, and

$$\dot{x}_i = Ax_i + Cz_i + Bu_i \tag{3.8}$$

if a vertex i is a leader agent. Here, $x_i \in \mathbb{R}^p$ is the state and $u_i \in \mathbb{R}^q$ is the input corresponding to each vertex, and z_i denotes the *coupling variable*, given by

$$z_i = K \sum_{(i,j)\in E} (x_j - x_i), \tag{3.9}$$

where $K \in \mathbb{R}^{s\times p}$ is the matrix describing the coupling strengths between the vertices. This means how strongly vertices affect each other, but it is not further treated in this paper. To put every single integrator system together, one can create the system

$$\dot{x} = (I_n \otimes A - L \otimes CK)x + (M \otimes B)u. \tag{3.10}$$

Here L is the Laplacian of the graph, K is as described in equation (3.9) and M describes which vertices have been assigned as leaders, so in this case is equal to U described in equation (3.5).

It is shown in [3, section II.B] that the controllability of this system shows the same properties as that of $\dot{x} = -Lx + Mu$. This result and the multi integrator dynamics style of equation (3.10) goes far beyond the scope of this paper, and therefore will not be studied extensively.

3.2 The zero forcing principle

After having introduced all preliminary topics, we can finally focus on the main problem of this paper. The question is for which possible leader sets, systems of the form (3.4) are controllable. To render such a system controllable, one has to find out which vertices have to be denoted as leader agents, thus which leader set is sufficient for controllability. In order to find a leader set for which (X, U) is controllable, one first has to know which properties this set must have.

To find a possible leader set, the *zero forcing principle* is introduced. In the first step, each vertex is given a colour: either black (corresponding to a leader agent) or white (corresponding to a follower agent). Once each vertex is assigned a colour, a process is started, in which more white vertices can be coloured black. This process consists of a few rules to be followed:

- A black vertex v can only colour a white vertex w black, if all vertices u for which $(u, v) \in E$ are all black as well. In other words, v must not have any adjacent white vertices aside from w.
- If a white vertex w can be coloured black, it is immediately coloured black.
- New black vertices are able to colour other white vertices black as well.
- Only white vertices can have their colour changed.
- If no other colourings are possible, the process terminates.

A Matlab code for this process can be found in appendix B, under the name `zeroforcing`.

The process of colouring white vertices black is also called *forcing* or *forcing black*. When a vertex v forces a vertex w to be black, for $v, w \in G$, we write $v \to w$. If the process of colouring vertices has ended, there are two possibilities:

1. The entire vertex set V has been coloured black
2. There are still some white vertices remaining

The most desirable case is to find a leader set V_L for which the entire vertex set can be forced black, as we will see further in this section.

Definition 3.2.1. A leader set $V_L \subseteq V$ for which the entire graph G can be forced black, is called a *zero forcing set* and is denoted by Z.

Example 3.2.1. Take the graph $G = (V, E)$ with $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1, 2), (2, 3), (2, 4), (3, 4), (3, 5), (4, 5), (4, 6), (5, 6)\}$. Such a graph is given in figure 3.2.
If we take $V_L = \{1, 4\}$ as in figure 3.2(a), the following steps can be made:

1. To begin with, since $d(1) = 1$, $1 \to 2$ immediately.
2. Then, vertex 2 only has 3 as adjacent white vertex, so $2 \to 3$
3. Lastly, both vertices 3 and 4 have one adjacent white vertex, namely 5. respectively 6, so these are both forced black.

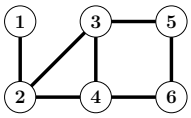


Figure 3.1: The graph $G = (V, E)$, as described in example 3.2.1

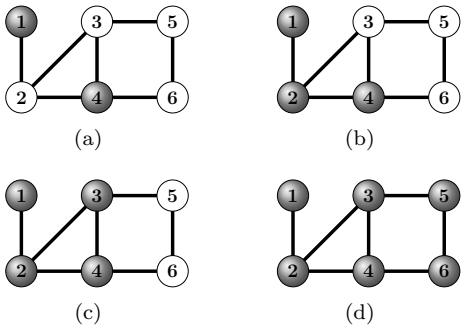


Figure 3.2: The zero forcing process of G with $V_L = \{1, 4\}$

Therefore, the leader set $V_L = \{1, 4\}$ is a zero forcing set of G. Since a leader set consisting of only one vertex cannot force the entire graph black, this is also a minimal leader set.

It is not said that any leader set V_L with $\text{card}(V_L) = 2$ also is a zero forcing set. If we take $V_L = \{1, 5\}$, it turns out that this is not a zero forcing set. Vertex 1 can force vertex 2 black, however, this is where the process terminates, see figure 3.3.

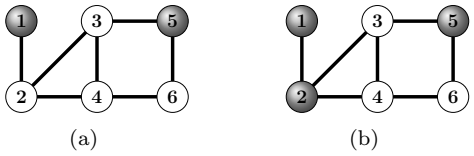


Figure 3.3: The zero forcing process of G with $V_L = \{1, 5\}$

Since we are interested in the controllability of systems defined on graphs, one may ask how the number of chosen leaders relates to controllability of the system. Together with the previous, one can observe the controllability of the pair (X, U) with U corresponding to a leader set V_L in the following way:

Theorem 3.2.1. (X, U) is controllable if and only if V_L is a zero forcing set.

Proof. This theorem is proven in [4, Page 6, Theorem 4.4] □

It is trivial that choosing the entire vertex set V as a leader set returns a zero forcing set, and renders (X, U) controllable. However, this is not the most ideal

case, since it is desirable to choose the least amount of leaders, that is, V_L has the lowest possible cardinality.

Definition 3.2.2. The *zero forcing number* $Z(G)$ of a graph G is the lowest cardinality of all possible zero forcing leader sets Z, i.e.

$$Z(G) = \min_{Z \subseteq V} |Z| \tag{3.11}$$

The first question that arises, is what number of leaders has to be chosen, in order to let the multi-agent system be controllable. Our main goal is to let the number of leaders be as few as possible. It is not trivial though, that any leader set V_L with $|V_L| = Z(G)$ is also a zero forcing set, see example 3.2.1. Therefore, one has to be very secure in the choice of leaders.

3.3 Relation to the eigenvalue problem

In order to determine a zero forcing set, and look whether it is optimal or not, it may be better if one first takes a look whether there exists a lower bound for the zero forcing number $Z(G)$, As controllability involves the rank of a matrix (see theorem 2.2.1), we are also able to relate the zero forcing problem to the minimal rank of a graph G.

In [6, section 1], it is described that out of a symmetric matrix $A \in \mathbb{R}^{n \times n}$, one can construct the *graph of a matrix* $\mathcal{G}(A)$. This graph containes the vertex set $V = \{1, \ldots, n\}$, so $|\mathcal{G}(A)| = n$ and the edge set $E = \{(i, j) : a_{ij} \neq 0,\ 1 \leq i < j \leq n\}$. Note that only the strict upper triangular part of the matrix A is concerned. Since we are still interested in undirected graphs, $a_{ij} = a_{ji}$ for each $i \neq j$, so the matrix A would be symmetric in any way.

Out of this, the *set of symmetric matrices of the graph* G is given by

$$S(G) = \{A \in S_n(\mathbb{R}) : \mathcal{G}(A) = G\}, \tag{3.12}$$

where $S_n(\mathbb{R})$ denotes the class of symmetric $n \times n$-matrices. Note that this set means exactly the same as the qualitative class of matrices of the graph G, as defined in equation 3.1. The *minimal rank* of the graph G is given by

$$\mathrm{mr}(G) = \min\{\operatorname{rank} A : A \in S(G)\}. \tag{3.13}$$

Additionally, the *maximal nullity* of the graph G is described by

$$\mathrm{M}(G) = \max\{N(A) : A \in S(G)\} \tag{3.14}$$

Theorem 3.3.1. $M(G) \leq Z(G)$

It is also mentioned in [6, Page 5] that for some studied graphs (at least with $|G| \leq 7$), it turns out that $M(G) = Z(G)$, although this is not proved explicitly. This result is used to check whether a certain found zero forcing set is actually

a minimal zero forcing set, and therefore, its cardinality is equal to the zero forcing number of the corresponding graph.

Since the minimal rank is not known explicitly for many graphs, we need another approach to obtain a lower bound for $Z(G)$. The Laplacian matrix L, however, can be determined for any graph G, as well as the geometric multiplicities of its eigenvales. Note that the *geometric multiplicity* of an eigenvalue $\lambda \in \sigma(L)$ is equal to the dimension of the eigenspace of λ corresponding to L, see [1, page 34].

Theorem 3.3.2. The zero forcing number $Z(G)$ is greater than or equal to the maximal geometric multiplicity of the eigenvalues of L. In other words, if Z is a zero forcing set for G, then:

$$\max_{\lambda \in \sigma(L)} \operatorname{mult} \lambda \leq Z(G) \leq |Z| \tag{3.15}$$

This result is also used for certain distance regular graphs. For those graphs, the maximal eigenvalue multiplicity can be calculated, and set as a lower bound for the zero forcing number $Z(G)$. In this way, one can check how efficient a certain chosen zero forcing set is. The results from theorems 3.3.1 and 3.3.2 will be used to verify whether the algorithms described in chapter 4 produce optimal sets.

We have to be a little careful there: if the cardinality of the found zero forcing sets are equal to the maximal eigenvalue multiplicity, then by theorem 3.3.2, this set has to be a minimal zero forcing set. However, if this is not the case, it is not set that this set is not minimal, since the maximal eigenvalue multiplicity is not necessarily equal to the zero forcing number! Only for graphs of which we will know for sure, the final zero forcing number will be shown.

3.4 Distance regular graphs

To define distance regular graphs, one needs the definition of a distance partition (see definition 2.1.1). Such a partition is of the form $\pi_D(v) = \{C_0, C_1 \ldots, C_{\text{diam}(G)}\}$, with initial vertex v, and each C_i contains all of the vertices $w \in V$ for which $\text{dist}(v, w) = i$.

Definition 3.4.1. [8, page 1] A connected graph G is called *distance regular* if for each $0 \leq i \leq \text{diam}(G)$ there exist $b_i, c_i \in \mathbb{N}$, such that for any two vertices $v, w \in V(G)$ with distance $\text{dist}(v, w) = i$, there are exactly c_i neighbours of v in $C_{i-1}(w)$ and exactly b_i neighbours of v in $C_{i+1}(w)$

These graphs have a specific structure that is highly uncommon for graphs in general, and there are several observations that have to be noticed:

- All vertices have the same degree (which is the definition of a *regular graph*). This degree is called the *valency* of the graph, and is denoted by k.
- The distance partition $\pi_D(v)$ has the same structure for each vertex v

Next to the numbers b_i and c_i, which we will refer to as the *forward degree* respectively *backward degree*, it may occur that two vertices in the same cell C_i are adjacent. This happens when $b_i + c_i < k$ for some $0 \leq i \leq \text{diam}(G)$. The *inner degree* a_i of the corresponding cell C_i is therefore given by

$$a_i = k - b_i - c_i. \tag{3.16}$$

The integers a_i, b_i and c_i are known as *intersection numbers* of the graph. Further in this paper, these numbers can be associated with corresponding cell C_i, but also with a vertex $v \in C_i$. All these numbers can be linked together in the following way:

Definition 3.4.2. [8, page 1] The *intersection array* of a distance regular graph G is given by

$$\{b_0, b_1, \cdots, b_{d-1}; c_1, \cdots, c_d\} \tag{3.17}$$

By definition, $b_0 = k$ and $c_0 = 1$ for each distance regular graph G. Notice that b_d and c_0 are not given in the intersection array, since the first and last cells of the distance partition do not have any backward respectively forward degrees.

Example 3.4.1. The *cubical graph* is distance regular. Its intersection array is given by $\{3, 2, 1; 1, 2, 3\}$. Note that the graphical representation of its distance partition indeed has the shape of a cube, see figure 3.4.

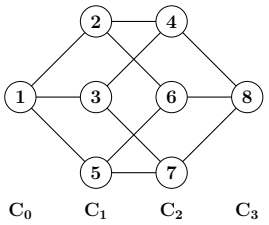


Figure 3.4: *The distance partition for the cubical graph, with initial vertex* $v = 1$.

The structure of the distance partition does not depend on its initial vertex. This can be graphically interpreted as choosing a vertex in a different corner to be the initial vertex, and therefore laying the cube on another side. This is to be verified by the reader.

Note that the cubical graph has the structure of a regular polytope. There are many distance regular graphs that correspond to regular polytopes, such as the *dodecahedral*, *icosahedral* and *sixteen-cell* graphs. Those graphs will be treated in this paper as well, and their results as regards to zero forcing sets, using the algorithms that are introduced in chapter 4 can be found in appendix A.

Definition 3.4.3. [8, page 128] The *standard sequence* $u_n(\lambda)$ corresponding to an eigenvalue λ of the Laplacian L, with $n = 0, \ldots, d$, is given iteratively by

- $u_{-1}(\lambda) = 0$,
- $u_0(\lambda) = 1$,

- $u_1(\lambda) = \frac{\lambda}{k}$,
- $c_i \cdot u_{i-1}(\lambda) + a_i \cdot u_i(\lambda) + b_i \cdot u_{i+1}(\lambda) = \lambda \cdot u_i(\lambda)$.

The standard sequence is used for determining the intersection numbers of a distance regular graph G.

Theorem 3.4.1. [9, section 12.2] The eigenvalues of the Laplacian L corresponding to the graph G are equal to the eigenvalues of the tridiagonal matrix

$$L_1 = \begin{bmatrix} a_0 & b_0 & 0 & \cdots & 0 \\ c_1 & a_1 & b_1 & \ddots & \vdots \\ 0 & c_2 & a_2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & b_{d-1} \\ 0 & \cdots & 0 & c_d & a_d \end{bmatrix}. \tag{3.18}$$

Theorem 3.4.2. [9, section 12.2] The dimension of the eigenspace corresponding to an eigenvalue λ of L is given by

$$\text{mult}(\lambda) = \frac{v}{\sum_{i=0}^{d} k_i \cdot u_i(\lambda)^2} \tag{3.19}$$

where v denotes the cardinality of the vertex set V and k_i the number of vertices in the corresponding cell C_i.

A Matlab script for computing these eigenvalue multiplicities, given a certain distance partition π_D of G, is given in in appendix B under the name `drgmulti`.

Chapter 4

Algorithms for finding zero forcing sets

The most straightforward way of finding $Z(G)$ is calculating each possible leader set, and check whether it is a zero forcing set or not. If all possible leader sets are found, look for the set that has the least amount of leaders (there may me more than one minimal set of leaders), and it is clear that its (their) number of leaders should equal $Z(G)$. An algorithm for this process can be found in appendix B under the name `zeroforcingpossibilities`.

The problem with this brute-force approach is that even for relatively small graphs, the computation speed exceeds the capacity of computers very quickly. For a graph with $|G| = n$, 2^n possible leader sets have to be checked on being a zero forcing set or not. As a result, the computational speed will explode when n increases.

Therefore, finding a zero forcing set of a graph needs a more sophisticated approach, which will be explained on the basis of a number of algorithms in this chapter. To begin with, we are able to give an upper bound for the zero forcing number for distance regular graphs.

Theorem 4.0.3. For the distance partition $\pi_D(v)$ of a distance regular graph G an upper bound for the zero forcing number Z is given by

$$Z(G) \leq 1 + \sum_{i=1}^{d}(|C_i| - 1) = n - \text{diam}(G) \tag{4.1}$$

This idea comes from the principle of a distance partition, together with the zero forcing principle, described in section 3.2. If one starts at cell C_0, which consists of only one vertex, and assume the following:

- The cell C_0 is forced black.
- If a cell C_i is forced black, then the next cell C_{i+1} needs to be forced black as well.

The last necessity is fulfilled from the fact that, once a cell is forced black and all previous cells have been forced black, and in the next cell there is only one

white vertex left (the $k_i - 1$-term in the summation in equation (4.1)), this one vertex needs to be forced black as well. As a consequence, the entire cell will be forced black, and inductively, the entire vertex set will be forced black. This principle will be used in the algorithms 2 and 3, described in the sections 4.2 respectively 4.3.
As one may have seen is that this approach is a very coarse one, and together with the eigenvalue multiplicies of a graph one may wonder whether there are no other options to produce upper bounds for the zero forcing number, together with a sufficient leader set V_L. In order to reach this desired upper bound, one can try to find zero forcing sets according to several algorithms. In this paper, a few algorithms are treated

4.1 Algorithm 1: the cutting edge algorithm

(This algorithm is found by N. Monshizadeh, MSc., PhD student in Systems and Control at the University of Groningen)

Description of Algorithm 1:

1. Determine for each vertex the *white degree*, that is, how many white vertices are adjacent to this vertex.
2. Take the lowest white degree, and call this number n. Assign this number of vertices (which have the lowest white degree) to be leaders, in the following way:
   (a) Assign such a vertex to be a leader, then the white degrees of adjacent vertices change.
   (b) Take the lowest white degree, and assign a corresponding vertex to be leader. Repeat this until n vertices have been chosen.
3. Determine which vertices can be forced black.
4. Vertices that have 0 as white degree must be chosen as leaders, as it is not likely they will be infected anymore.
5. If the determination gets stuck, repeat the entire process, until the complete vertex set V has been forced to zero.

Note that this algorithm is not necessarily focused on distance regular graphs, but any (undirected) graph in general. Therefore, the next example does not concern a distance regular graph, but a graph that is meant to emphasize the operation of this algorithm.

Example 4.1.1. Consider the graph $G = (V, E)$, with $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ and $E = \{(1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (7, 8), (7, 9), (8, 10), (9, 10)\}$, see figure 4.1. The zero forcing algorithm works as follows:

1. To begin with, each vertex is white. The vertices $\{2, 3, 4, 5, 6\}$ clearly have the lowest white degree.
2. Only one leader has to be chosen, so we take 2 as a new leader.

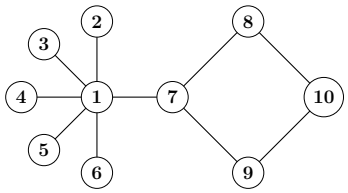


Figure 4.1: Graph example for algorithm 1

3. Thus, $2 \to 1$. Here terminates the zero forcing process.
4. The vertices $\{3, 4, 5, 6\}$ have white degree 0, and are still white. Since they have the lowest white degree, they must be chosen as leaders.
5. From here, $1 \to 7$. Now the process terminates again
6. The vertices 8 and 9 now have the lowest white degree (which is 1), and either one of them has to be chosen as a leader. We choose 8 to be a new leader.
7. Now, $7 \to 9$ and $8 \to 10$. Therefore, the entire graph is forced black.

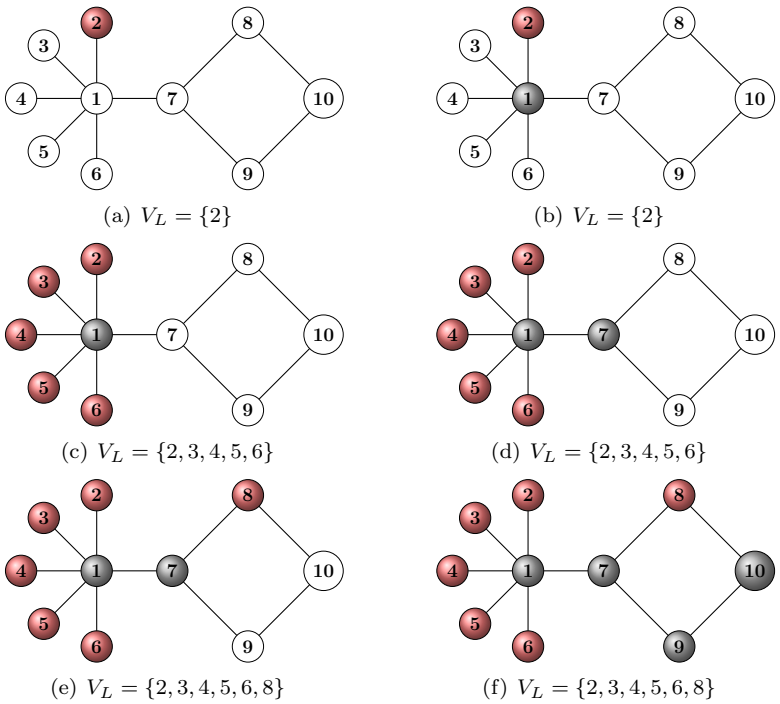


Figure 4.2: The cutting edge algorithm applied on the graph G of figure 4.1

The found leader set is $V_L = \{2, 3, 4, 5, 6, 8\}$, and consists of six leaders. However, this does not imply that this is actually the zero forcing number! Using

the script `zeroforcingpossibilities`, it turns out that the smallest possible leader set indeed consists of six leaders. Therefore, this algorithm produces a desired result for several classes of graphs.

4.2 Algorithm 2: the distance partition algorithm

(This algorithm is found by M.K. Camlibel, PhD and professor in Systems and Control at the University of Groningen)

Description of Algorithm 2:

1. Determine the distance partition for a given initial vertex
2. Assign the only vertex in the first cell to be a leader.
3. Assume the i-th cell to be forced black, then look at how the $i+1$-th cell can be forced black as well in the following way:
   (a) Compute the so-called *forward white degree* for each vertex in the cell. If the property is, all these vertices have the same forward degree (because of distance regularity). Name it p.
   (b) Take $p-1$ vertices in the next cell as leaders (unless $p=1$, then take only one), then determine which vertices are forced black.
   (c) The forward white degrees may have changed now, take the least one and name it p again. turn another $p-1$ vertices into leaders.
   (d) Repeat this process, until the $i+1$-th cell has completely been forced black.
4. Inductively, the entire graph is forced black in this way.

One can imagine that the sophisticated approach of this algorithm produces a leader set with smaller cardinality than the rough method according to equation (4.1), that is furthermore chosen in a more clever way.

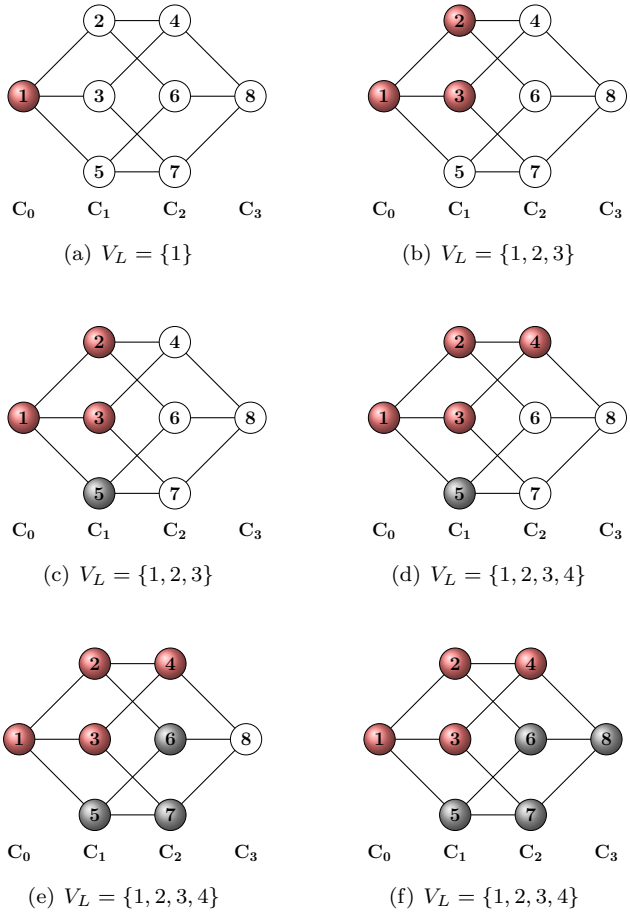
Example 4.2.1. As mentioned in section 3.4, the cubical graph is distance regular. We are now trying to find a leader set V_L that forces this graph black. According to the algorithm, we perform the following steps:

1. The distance partition of this graph has already been determined, see figure 3.4. Since we chose vertex 1 to be the initial vertex, we assign this vertex as a leader. Therefore, C_0 is forced black immediately (figure 4.3(a)).
2. Take a look at C_1. Since $b_0 = 3$, we have to two vertices in C_1 as a leader, say 2. The leader set is now $V_L = \{1, 2, 3\}$, which does not force any other vertices (figure 4.3(b)).
3. Both vertices 3 and 5 still have a forward white degree of 2. Therefore, we assign again one of them as a leader, say 3. This causes vertex $1 \to 5$, and therefore, C_1 has completely been forced black (figure 4.3(c)).

4. Now look at C_2. Each vertex has forward white degree of 1, so we only have to assign one additional vertex as a leader. For instance, take 4 (figure 4.3(d)).

5. Consequently, $2 \to 6$ and $3 \to 7$. By now, C_2 has been entirely forced black (figure 4.3(e)).

6. Since each vertex in C_2 all have only one white adjacent vertex, which is 8, this vertex is infected automatically. Therefore, the entire graph is forced black. The resulting zero forcing set is $Z = \{1, 2, 3, 4\}$ (figure 4.3(f)).



(a) $V_L = \{1\}$ (b) $V_L = \{1, 2, 3\}$

(c) $V_L = \{1, 2, 3\}$ (d) $V_L = \{1, 2, 3, 4\}$

(e) $V_L = \{1, 2, 3, 4\}$ (f) $V_L = \{1, 2, 3, 4\}$

Figure 4.3: The distance partition algorithm applied on the cubical graph

It is given in [6, Page 3] that the minimal rank for an n-dimensional hypercube Q_n is equal to 2^{n-1}. Since $n = 3$ for the cubical graph, the minimal rank, and therefore the zero forcing number, is $Z(G) = 4$. Since this is exactly the number of leaders we have found with this algorithm, according to theorem 3.3.1, we know that this is number is actually the optimal number of leaders.

The main advantage of this method is that one uses the local structure of the distance partition in a very efficient way. However, the algorithm does not take

account of the further structure of the graph; the iteration process only uses information of two cells at once. In this way, useful information that is available in further cells (cells C_j in relation to the current cell C_i, where $j-1 > i$) is not taken account of, and when the previous cells have already been forced black, will perhaps never be taken account of.

4.3 Algorithm 3: the path algorithm

In order to keep the further structure of the distance regular graph G in mind, an additional algorithm for finding a possible leader set V_L is introduced, making use of paths inside the distance partition $\pi_D(G)$ from one side (cell C_0) to the other (cell $C_{\mathrm{diam}(G)}$). In this way, both the usage of the local structure between two cells C_i and C_{i+1}, and the further structure of the graph G, is conserved.

Description of Algorithm 3:

1. Determine the distance partition for a given initial vertex
2. Assign the only vertex in the first cell to be a leader.
3. Create a path of black vertices from one side of the distance partition to another, in the following way:
   (a) Start at the initial vertex (which is the only vertex in the initial cell)
   (b) Take a white vertex from the next cell, assign it as a leader, and look whether other vertices can be forced black.
   (c) If there is a black vertex in the next cell, jump to the cell after that one, continuing from the concerned black vertex. If not, assign a white vertex in the next cell as a leader vertex, and continue from there.
   (d) Go on this way, until a path of black vertices is created from the other side of the distance partition to the other.
4. Create new paths paths from black forced cells, recursively, in the following way:
   (a) Start in the last cell that has entirely been forced black.
   (b) Take the first vertex in the cell (from above) that has no black neighbours in the next cell.
   (c) Repeat the same process as in the previous step, until a path of black vertices from one side to the other is reached.
5. Keep creating paths, until the entire vertex set is forced black.

Example 4.3.1. The icosahedral graph, which again corresponds to a regular polytope (the icosahedron), can be represented as in figure 4.4. According to the path algorithm, we create a zero forcing set for this graph in the following way:

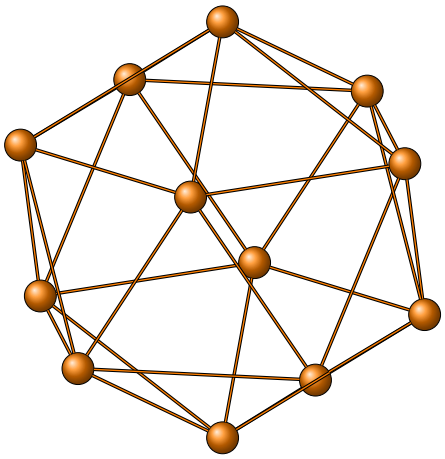


Figure 4.4: The icosahedral graph

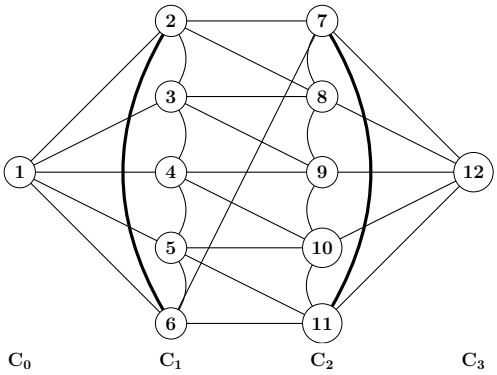


Figure 4.5: The distance partition for the icosahedral graph

1. We construct a distance partition with initial vertex 1. It is to be verified to the reader that the graph in figure 4.5 is similar to the graph in figure 4.4

2. We set vertex 1 as a leader. No other vertices are infected, though C_0 is now completely black (figure 4.6).

3. Now we start creating a path of black vertices from C_0 to C_3. First, we assign a vertex in C_1 as a leader, say 2 (figure 4.7).

4. This new leader does not cause any other vertex to be infected. Therefore, continue the path towards vertex 12. Assign vertices 7 and 12 as leaders. Again, this combination of leaders does not force any other vertices black (figure 4.8).

5. Now that the first path has been created, we create a path from 1 in another direction. The most obvious choice is vertex 3 (figure 4.9).

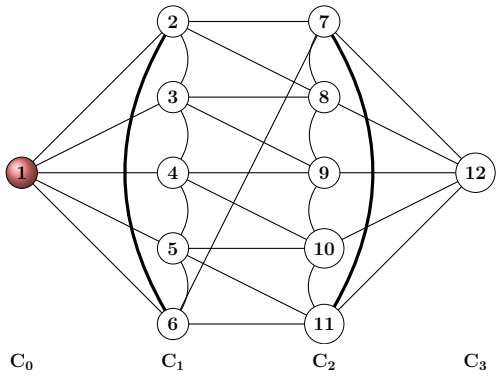


Figure 4.6: $V_L = \{1\}$.

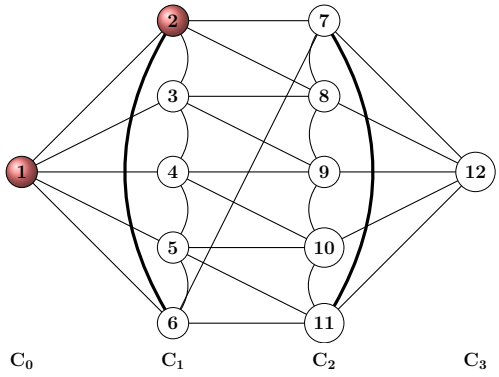


Figure 4.7: $V_L = \{1, 2\}$.

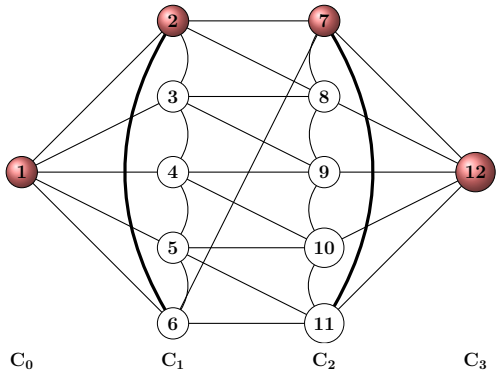


Figure 4.8: $V_L = \{1, 2, 7, 12\}$.

6. So far, no infections have occurred. Although, with our next choice, the entire graph will be forced black. Take 8 as a new leader, then $2 \to 6$ and $8 \to 9$ (figure 4.10).

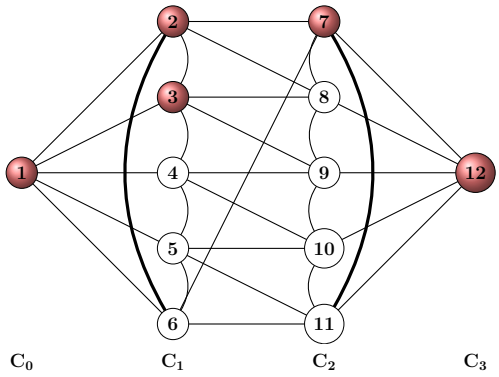


Figure 4.9: $V_L = \{1, 2, 3, 7, 12\}$.

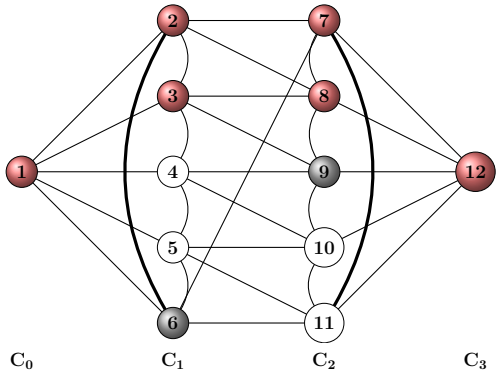


Figure 4.10: $V_L = \{1, 2, 3, 7, 8, 12\}$.

7. Now, $3 \to 4$ and $9 \to 10$ (figure 4.11).

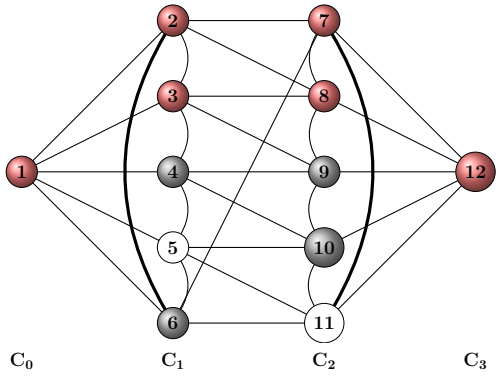


Figure 4.11: $V_L = \{1, 2, 3, 7, 8, 12\}$.

8. Lastly, 5 and 11 are infected (possible from many directions). Therefore,

the whole graph has now been forced black (figure 4.12).

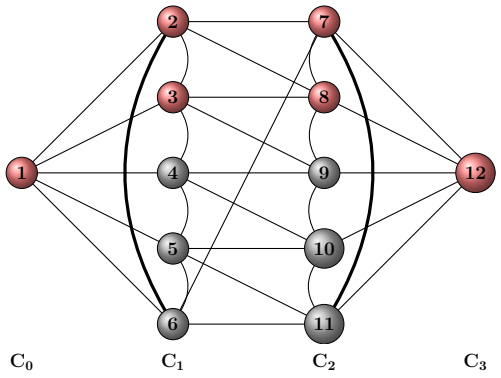


Figure 4.12: $V_L = \{1, 2, 3, 7, 8, 12\}$.

4.4 Results for several distance regular graphs

Now that we have introduced the algorithms, it is time to compare them for several known examples of distance regular graphs. It turns out that not every algorithm produces the same amount of leaders for a certain graph, and it is not always the same algorithm that produces the least amount of leaders. Therefore no single algorithm turns out to give an optimal solution. The results for those graph can be found in appendix A.

We now show the processes described in section 4.1 applied on a few graphs that give a remarkable difference between these algoritms.

Example 4.4.1. The Heawood graph is given in figure 4.13

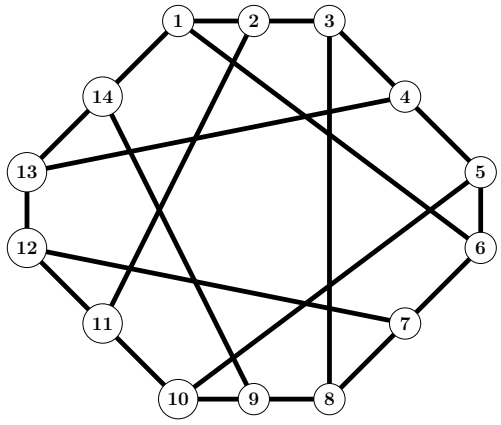


Figure 4.13: The Heawood graph

The striking result will be that algorithm 1 will produce a leader set with a

cardinality of 6, whereas algorithm 2 produces one with a cardinality of 7. Therefore, we will look at both processes. To begin with, we take algorithm 1:

1. The white degree of each vertex is equal to 3, so it does not matter which vertex we assign as leader. Without loss of generality, we take vertex 1 as a leader (figure 4.14).

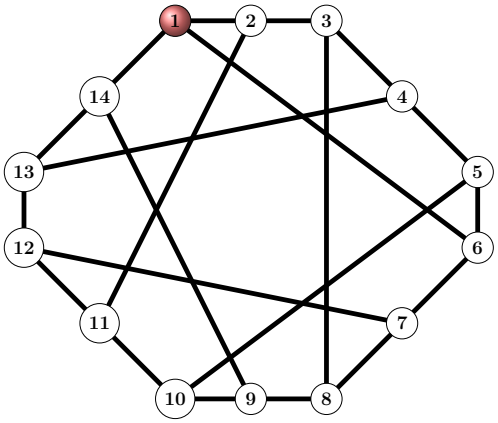


Figure 4.14: $V_L = \{1\}$.

2. Two additional vertices have to be assigned as leaders, since this will cause another vertex to be forced black. The white degrees of vertices 2, 6 and 14 have changed, so we have to take one of them as a leader, say 2 (figure 4.15).

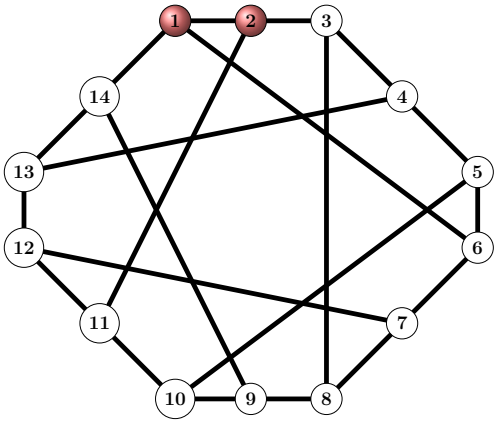


Figure 4.15: $V_L = \{1, 2\}$.

3. Now, the white degrees of the vertices 3 and 11 have lowered by one. Take 3 as an additional leader, then we look which vertices are forced black (figure 4.16).

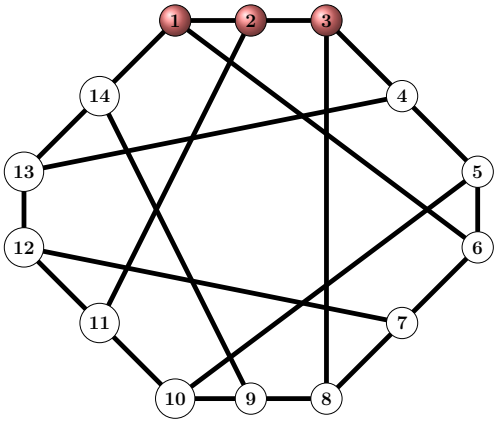


Figure 4.16: $V_L = \{1, 2, 3\}$.

4. Now $2 \to 11$, by choosing 1 and 3 as leaders. This is the only infection that can be done so far (figure 4.17).

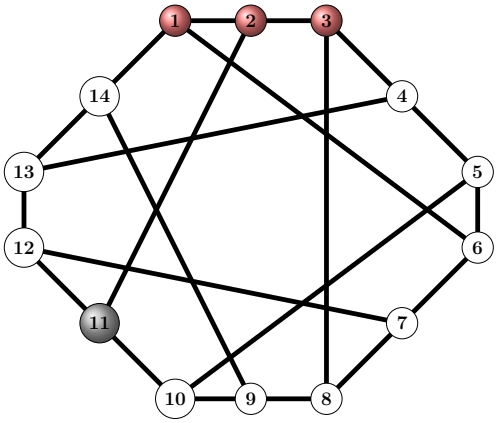


Figure 4.17: $V_L = \{1, 2, 3\}$.

5. Now, we have to pick an additional collection of leaders. The white degrees of vertices 4, 8, 10, 12, and 14 are the lowest, so choose for instance 4 to be a leader. This causes $3 \to 8$ (figure 4.18).

6. The white degrees of the vertices 3 and 11 have lowered by one, but those of the preceding vertices have not changed yet. Therefore, pick 5 as a leader, so $4 \to 13$ (figure 4.19).

7. Analogously, we choose 6 to be a leader. Since $5 \to 10$, $6 \to 7$, $8 \to 9$, $11 \to 12$ and $13 \to 14$, this is the last choice we have to make (figure 4.20).

Now that we have established the leader set, we now take a look at algorithm 2. Therefore, we need the distance partition of the Heawood graph, which is

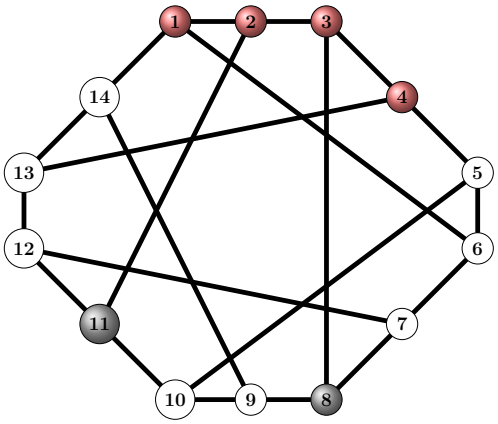


Figure 4.18: $V_L = \{1, 2, 3, 4\}$.

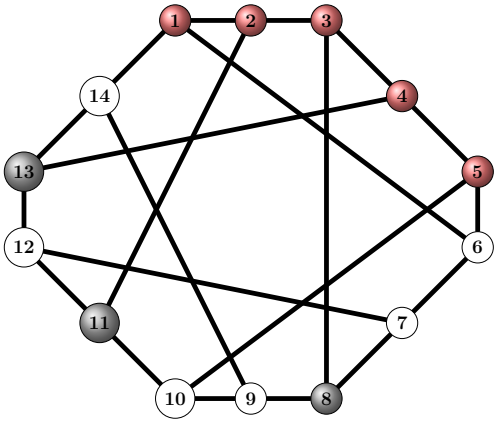


Figure 4.19: $V_L = \{1, 2, 3, 4, 5\}$.

given in figure 4.21. It is to be verified by the reader that this distance partition corresponds to the graph in figure 4.13. Algorithm 2 is used in the following way:

1. First, we take the initial vertex of the distance partition, 1, as a leader (figure 4.22).
2. Since the forward white degree of vertex 1 is equal to 3, we have to take two vertices in order to force C_1 black. Therefore, we take vertices 2 and 6 as leader, so $1 \to 14$ (figure 4.23).
3. The minimal forward white degree of each vertex in C_1 is two. Therefore, we only have to pick one vertex to infect any vertices in C_2, say vertex 3. This causes $2 \to 11$ (figure 4.24).
4. Since vertices 3 and 11 are not adjacent to 6 or 14, the white degree of these last two vertices remains equal to 2. Analogously, vertex 5 is chosen as a leader, so $6 \to 7$, and 9 is chosen, so $14 \to 13$ (figure 4.25).

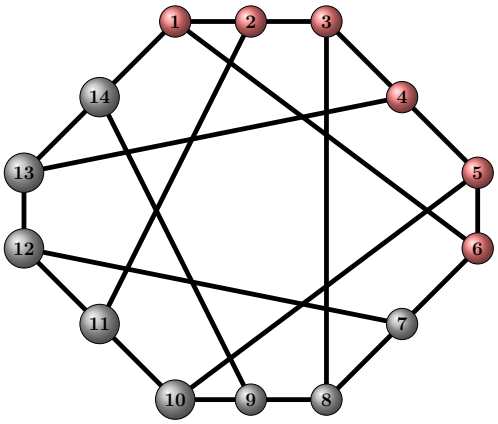


Figure 4.20: $V_L = \{1, 2, 3, 4, 5, 6\}$.

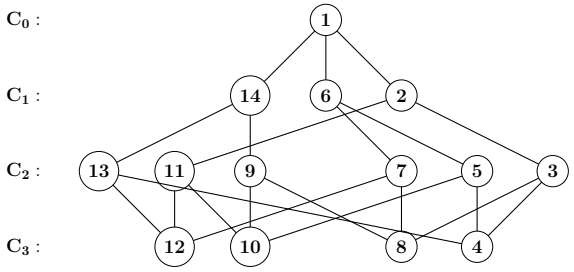


Figure 4.21: The distance partition for the Heawood graph.

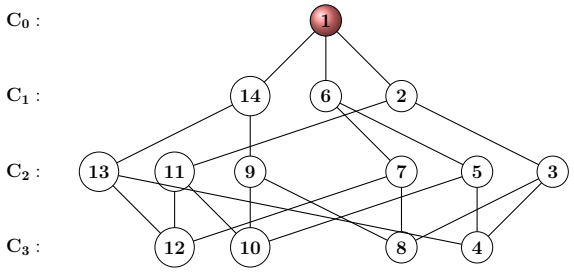


Figure 4.22: $V_L = \{1\}$.

5. By now, we have chosen six leader, though not all vertices have been forced black. To finalize this example, we choose 4 as a leader, which causes the remainder vertices to be black (figure 4.26).

Since we had to choose more than six vertices to form a zero forcing set, this algorithm is less efficient for the Heawood graph than algorithm 1. This occurs since the algorithm only makes use of two cells at once, and does not interfere the remainder structure of the graph during the process of infecting vertices in

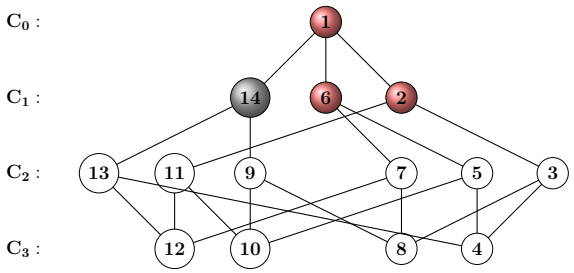


Figure 4.23: $V_L = \{1, 2, 6\}$.

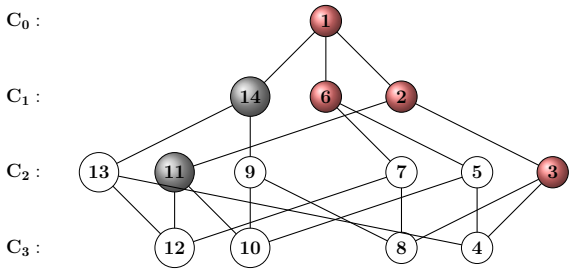


Figure 4.24: $V_L = \{1, 2, 3, 6\}$.

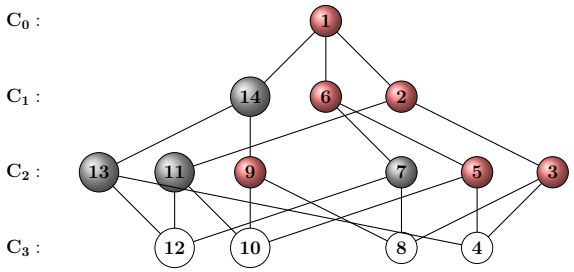


Figure 4.25: $V_L = \{1, 2, 3, 5, 6, 9\}$.

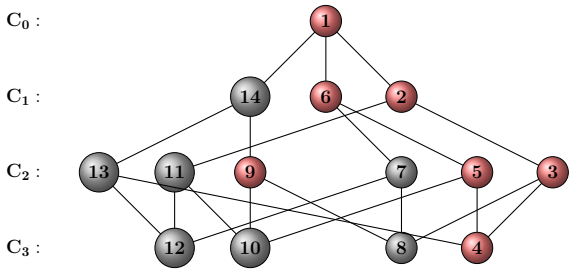


Figure 4.26: $V_L = \{1, 2, 3, 4, 5, 6, 9\}$.

the currrent cell. It can be shown that algorithm 3, which does make use of the further structure, indeed produces a leader set that consists of six vertices. This however, is to be determined by the reader.

The most remarkable example is the Icosahedral graph, on which the third algorithm (creating paths from one side of the distance partition to another) will be demonstrated. While creating a distance partition, it does matter in which cell the vertices are placed, as this determines the distance to the initial vertex.

Initially, it does not seem to matter in which order the vertices are place. However, it does matter when the algorithm depends on the order of the vertices (from up to down) inside a cell C_i, and that is the case with algorithm 3. The next example has to make clear why this order is that important.

Example 4.4.2. A distance partition of the Icosahedral graph, as explained in example 4.3.1, is given in figure 4.27. The vertices are though placed in a different order within the cells.

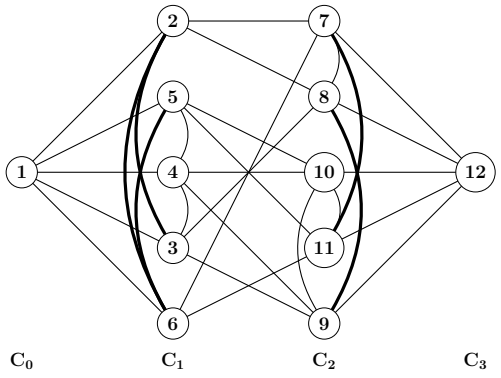


Figure 4.27: Another distance partition for the icosahedral graph

It is to be verified to the reader that the partition in figure 4.27 equals that of figure 4.5. Now we apply the path algorithm to this distance partition.

1. We set vertex 1 as a leader. No other vertices are infected, though C_0 is now completely black (figure 4.28).

2. As mentioned in example 4.3.1, we start creating a path of black vertices from C_0 to C_3. In order to reach that, again we choose 2, 7 and 12 as leaders. The leaders on itself will not infect any other vertices, so the situation of example 4.3.1 remains the same (figure 4.29).

3. This is the point where the first differences occur. According to algorithm 3, we have to create a new path from 1 to 12, so vertex 5 has to be chosen as a leader, although this may not be the wisest decision (figure 4.30).

4. No other vertices are infected, so we continue creating the path. Choose vertex 10 as an additional leader. Again, no other vertices are forced black (figure 4.31).

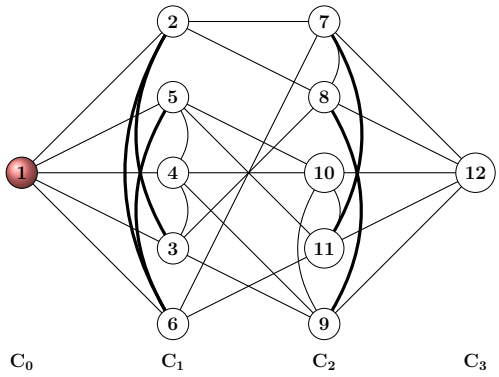


Figure 4.28: $V_L = \{1\}$.

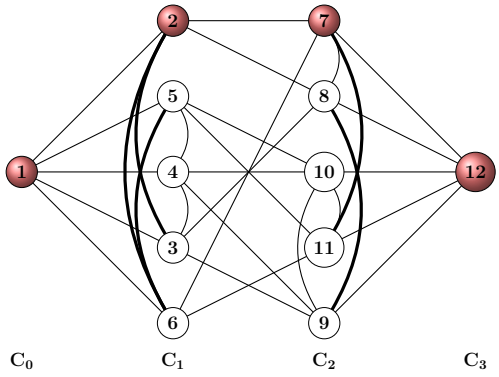


Figure 4.29: $V_L = \{1, 2, 7, 12\}$.

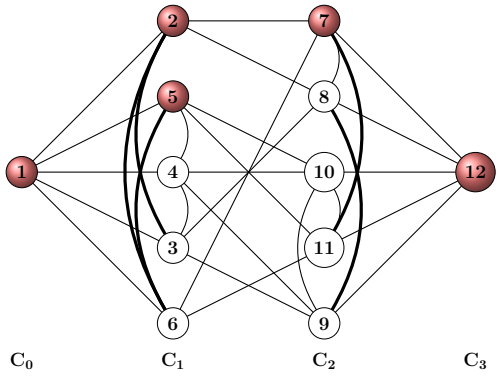


Figure 4.30: $V_L = \{1, 2, 5, 7, 12\}$.

5. This is the main difference compared to the former distance partition, which has been used in example 4.3.1, where six leaders were enough to force the entire graph black. Here, we have already chosen six leaders,

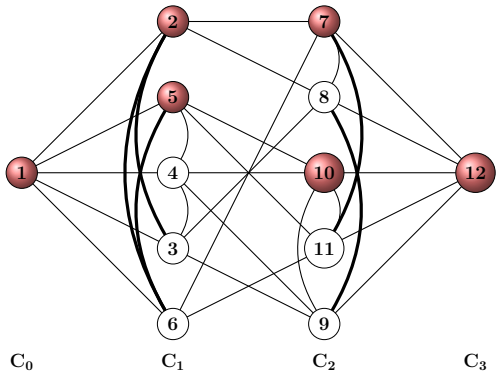


Figure 4.31: $V_L = \{1, 2, 5, 7, 10, 12\}$.

and none of the other vertices have been forced black, though we used the same algorithm.

To make this example complete, we create an additional path by picking 4 as a leader (figure 4.32).

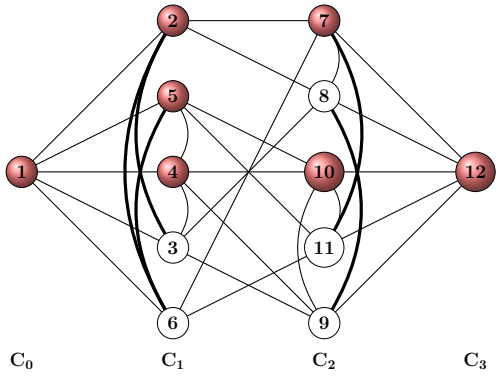


Figure 4.32: $V_L = \{1, 2, 4, 5, 7, 10, 12\}$.

6. This does not have any result with respect to the other white vertices. Next, pick 9 as a leader, which is the first leader that causes a vertex to turn black (figure 4.33).

7. Consequently, $4 \to 3$ and $10 \to 11$ (figure 4.34).

8. Ultimately, vertices 6 and 8 are forced. Therefore, the entire graph has now been forced black (figure 4.35).

It is now clear that the choice of the distance partition does affect the cardinality of the chosen leader set. A wise choice of the order of the vertices inside a cell does sometimes make a difference.

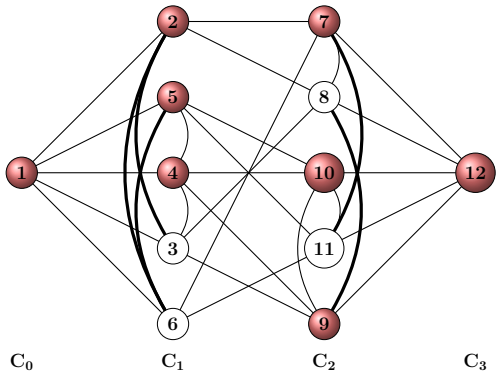


Figure 4.33: $V_L = \{1, 2, 4, 5, 7, 9, 10, 12\}$.

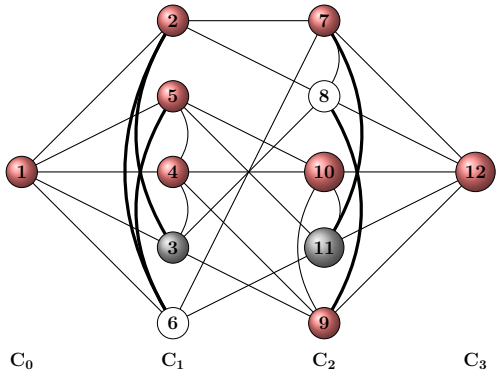


Figure 4.34: $V_L = \{1, 2, 4, 5, 7, 9, 10, 12\}$.

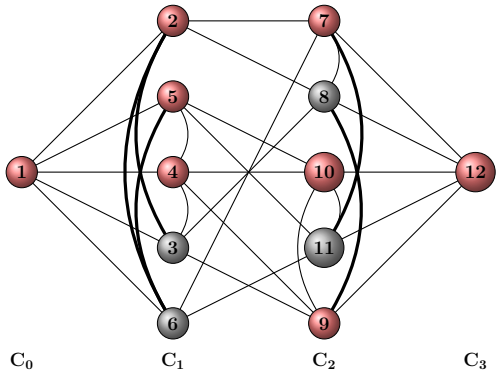


Figure 4.35: $V_L = \{1, 2, 4, 5, 7, 9, 10, 12\}$.

Given a cell C_i for which the vertices are determined correctly (those with $\text{dist}(v, w) = i$, where v is the initial vertex and $w \in C_i$), but have not been put in the correct order yet. It turns out that the best choice is to put vertices, that

are adjacent to the most upper vertex in cell C_{i-1}, also on top of C_i, then those that are adjacent to the second upper vertex in C_{i-1}, and so on. An algorithm for creating such distance partition as in figure 4.5 is given in appendix B.

Chapter 5

Conclusion

In this thesis, we defined a class of linear systems on graphs, in order to be able to define our problem with the assigning of leader agents. Then, we introduced the zero forcing principle, and leader sets that fulfilled the prerequisites of this principle turned out to be a sufficient leader set, speaking of controllability of the system.

As mentioned in chapter 1, our main goal was to find a leader set for which a linear multi-agent system is controllable, and the cardinality of the leader set is still minimal. The number of leaders for which a zero forcing leader set is minimal was referred to as the zero forcing number. Since there was no straightforward way of determining a minimal leader set, we had to look whether this number can be determined using several properties of graphs, such as minimal rank and maximal eigenvalue multiplicites.

The main class of graphs treated in this thesis was that of distance regular graphs, and it turned out that their maximal eigenvalue multiplicity was relatively easy to calculate in order to determine their zero forcing numbers. This was useful in chapter 4, where several algorithms for finding zero forcing sets were introduces, especially for distance regular graphs. By using these algorithms, we could find reasonable leader sets for some known distance regular graphs.

As promised in chapter 1, some surprising results have come up, concerning the effectivity of the algorithms on different graphs. To begin with, not all the algorithms always produces the optimal leader set, and it is not always the same algorithm that produces the least number of leaders for each graph. For instance, the distance partition algorithm (algorithm 2, described in section 4.2), works the best in a lot of cases, except for example the Heawood graph.

The most subtle case was the Icosahedral graph, for which it made sense in which way the distance partition was produced. It turned out by the way that this was the only tested graph for which this was the case; all the other graphs always obtained the same number of leaders with the path algoritm (algorithm 3, described in section 4.3). This will undoubtedly be a point of further research, since there may be other examples for which this is the case.

There is another idea for further research we have to focus on for a while. In the path algorithm, the distance partition is forced black from the left (the earlier cells) to the right (the latter cells). It could be possible to create paths from up (the first vertices in each cell) to down (the latter vertices in each cell). This will not further be treated in this paper anymore, though it may be a point of research.

The last thing that has to be mentioned, is that the cutting edge algorithm (algorithm 1, described in section 4.1) is not especially designed for distance regular graphs. Therefore, it can be used on many other classes of graphs, thinking of *regular graphs* and all of its deviations, that can be found in [8, chapter 1].

At the beginning of this thesis, the problem seemed relatively easy. However, when we had a solution to the problem, we could not even verify whether this was an optimal solution. By introducing several tools, we managed to determine the zero forcing number for some graphs indeed, and ultimately solve the zero forcing problem for those graphs!

Bibliography

[1] G.J. Olsder, J.W. van der Woude, *Mathematical Systems Theory*, 3rd edition, VSSD, Delft, 2005.

[2] M. Mesbahi, M. Egerstedt, *Graph Theoretic Methods in multiagent Networks*, Princeton University Press, 2010.

[3] S. Zhang, M. Cao, M.K. Camlibel, *Upper and lower bounds for controllable subspaces of networks of diffusively coupled agents*, IEEE Transactions on Automatic Control, Volume: PP, Issue: 99, 2013.

[4] N. Monshizadeh, S. Zhang, M.K Camlibel, *Zero forcing sets and controllability of dynamical systems defined on graphs*, IEEE Transactions on automatic control (Volume: PP ,Issue: 99), 2013

[5] L. Hogben, *Minimum rank problems*, Linear Algebra and its Applications 432, pp 1961-1974, 2010.

[6] AIM Minimum Rank - special graphs work group, *Zero forcing sets and the minimum rank of graphs*, Linear Algebra and it Applications. 428, pp 1628-1648, 2008

[7] D.M. Cardoso, C. Delorme, P. Rama, Laplacian eigenvectors and eigenvalues and almost equitable partitions, *European Journal of Combinatorics*, 28:665-673, 2007.

[8] A.E. Brouwer, A.M. Cohen, A. Neumaier, *Distance Regular Graphs*, Springer, 1989.

[9] A.E. Brouwer, W.H Haemers, *Spectra of Graphs*, Springer, 2011.

Appendix A

Table with results for distance regular graphs

Name:	**Number of vertices**	**Number edges**	**Intersection array**
Biggs-Smith	102	153	$\{3,2,2,2,1,1,1;1,1,1,1,1,1,3\}$
Brouwer-Haemers	81	810	$\{20,18;1,6\}$
Clebsch	16	40	$\{5,4;1,2\}$
Coxeter	28	42	$\{3,2,2,1;1,1,1,2\}$
Cubical	8	12	$\{3,2,1;1,2,3\}$
Desargues	20.	30	$\{3,2,2,1,1;1,1,2,2,3\}$
Dodecahedral	20	30	$\{3,2,1,1,1;1,1,1,2,3\}$
Foster	90	135	$\{3,2,2,2,2,1,1,1;1,1,1,2,2,2,2,3\}$
Hadamard-8	8	6	$\{2,1,1,1;1,1,1,2\}$
Hadamard-32	32	128	$\{8,7,4,1;1,4,7,8\}$
Hadamard-48	48	288	$\{12,11,6,1;1,6,11,12\}$
Hadamard-64	64	512	$\{16,15,8,1;1,8,15,16\}$
Heawood	14	21	$\{3,2,2;1,1,3\}$
Icosahedral	12	30	$\{5,2,1;1,2,5\}$
Klein	24	84	$\{7,4,1;1,2,7\}$
Levi	30	45	$\{3,2,2,2;1,1,1,3\}$
Octahedral	6	12	$\{4,1;1,4\}$
Pappus	18	27	$\{3,2,2,1;1,1,2,3\}$
Pentatope	5	10	$\{4;1\}$
Petersen	10	15	$\{3,2;1,1\}$
Shrikhande	10	48	$\{6,3;1,2\}$
Sixteen Cell	8	24	$\{6,1;1,6\}$
Tesseract	16	32	$\{4,3,2,1;1,2,3,4\}$
Tetrahedral	4	6	$\{3;1\}$
Tutte 12-Cage	126	189	$\{3,2,2,2,2,2;1,1,1,1,1,3\}$
Utility	6	9	$\{3,2;1,3\}$
Wells	32	80	$\{5,4,1,1;1,1,4,5\}$

Name:	**Algorithm 1**	**2**	**3**	**Maximum Eigenvalue Multiplicity**	**Known $Z(G)$**
Biggs-Smith	21	22	21	18	
Brouwer-Haemers	79	67	67	60	
Clebsch	12	10	10	10	$\Rightarrow 10$
Coxeter	10	10	9	8	
Cubical	4	4	4	3	4
Desargues	10	8	8	5	
Dodecahedral	6	6	6	5	
Foster	25	27	22	18	
Hadamard-8	2	2	2	2	$\Rightarrow 2$
Hadamard-32	30	22	26	14	
Hadamard-48	46	36	40	22	
Hadamard-64	62	52	56	30	
Heawood	6	7	6	6	$\Rightarrow 6$
Icosahedral	11	6	8	5	
Klein	23	15	15	8	
Levi	10	13	11	10	$\Rightarrow 10$
Octahedral	5	4	4	3	
Pappus	8	7	7	6	
Pentatope	4	4	4	4	$\Rightarrow 4$
Petersen	5	5	5	5	$\Rightarrow 5$
Shrikhande	15	10	10	9	
Sixteen Cell	7	6	6	4	
Tesseract	8	8	8	6	
Tetrahedral	3	3	3	3	$\Rightarrow 3$
Tutte 12-Cage	29	49	32	28	
Utility	4	4	4	4	
Wells	25	20	16	10	

Appendix B

MATLAB codes for the given algorithms

adjacencymatrix

```
function [A] = adjacencymatrix(V,E)

% Adjacency Matrix
%
% Input variables:
%
%   V:  Number of vertices of the graph G (natural number)
%   E:  Edge set of the graph G (n x 2-matrix)
%
% Output variables:
%
%   A:  Adjacency matrix of the graph (n x n-matrix)
%
% This function is supposed to determine adjacency matrix for a given
% undirected (!) graph G(V,E), returned as an n x n-matrix.

% Preallocation of variables

numberofedges = size(E,1)
A = zeros(V)

for edge = 1:numberofedges
    A(E(edge,1),E(edge,2)) = 1
    A(E(edge,2),E(edge,1)) = 1
end

end
```

cuttingedgealg

```
function [numberofleaders] = cuttingedgealg(A)

% CUTTINGEDGEALG
%
% Input Variables:
%   V: number of vertices of G (natural number)
%   E: edge set of G (n x 2-matrix)
%
% Output Variables:
%   leaderset: The leader set found accoring to the given algorithm
%   (V-dimensional row vector)

V = size(A,1)
E = edgeset(A)

if V >= 1 && round(V) == V && size(E,2) == 2
else
    disp('The variables do not satisfy the given conditions')
    return
end

% Nima's Algorithm for finding a zero forcing set for a given graph G(V,E):
%
% 1.    Determine, for each vertex, the "white degree"
% 2.    Take the lowest white degree, assign this number of vertices (which
%       have the lowest white degree) to be leaders, in the following way:
%       - Assign such a vertex to be a leader, then the white degrees of
%         adjacent vertices change.
%       - Take the lowest white degree, and assign a corresponding vertex
%         to be leader. Then repeat this process, until n vertices have
%         been chosen.
% 3.    Determine which vertices can be forced in the next steps.
% 4.    If the determination gets stuck, repeat the entire process, until
%       the complete vertex set V has been forced to zero.

% Preallocation of variables:

whitevertices = ones(1,V);
blackvertices = zeros(1,V);
leaderset = zeros(1,V);

% Initially, the number (or set) of white vertices is equal to the number
% (or set) of given vertices V in the graph G(V,E). When a white vertex is
% changed to black, the corresponding entry will be changed to 0.

% If this one is also chosen as additional leader, the corresponding entry
% in "leaderset" will be changed to one. The idea is to let "whitevertices"
% and "blackvertices" be each others complement, and building up the leader
```

```
% set separately.

while nnz(blackvertices) ~= V

% Step 1:

% Determine the white degree for each white vertex:

whitedegrees = whitedegree(V,whitevertices,E)

% Do not confuse the row vector "whitedegrees" (left) and the function
% "whitedegree" (right).
%
% The white degrees have been determined, now we have to look at the
% minimal one (note that the white degrees for black vertices and
% "nonadjacent" vertices cannot be chosen, unless the minimal white degree
% equals V, but this is very unprobable).

minimalwhitedegree = min(whitedegrees)

minimalvertices = find(whitedegrees == minimalwhitedegree)

% If there is a black vertex involved, the corresponding white degree is
% set very high, so black vertices will never be taken as minimal.

whitevertices(minimalvertices(1)) = 0
leaderset(minimalvertices(1)) = 1
blackvertices(minimalvertices(1)) = 1

% Until an additional chosen leader has a white degree of 1, it makes sense
% to check whether additional vertices are forced black. If the minimal
% white degree of all white vertices is still greater than one, running
% zeroforcing does not help.


    blackvertices = zeroforcing(V,leaderset,E)
    whitevertices = ones(1,V) - blackvertices


if nnz(blackvertices) == V
    break
end

% It is possible that there are vertices with a white degree of 0, but are
% not forced black. It is completely necessary to assign those vertices as
% leaders, because they are not going to be forced black in any way!

whitedegrees = whitedegree(V,whitevertices,E)
zerodegrees = find(whitedegrees == 0)
if numel(zerodegrees) == 0
```

```
else
    if minimalwhitedegree == 1
    leaderset(zerodegrees) = 1
    whitevertices(zerodegrees) = 0
    blackvertices(zerodegrees) = 1

    blackvertices = zeroforcing(V,leaderset,E)
    whitevertices = ones(1,V) - blackvertices
    end
end




% In the end, the computed (black) leader set is equal to the set of
% vertices, without the set of white vertices.

end

numberofleaders = nnz(leaderset);

leaderset;

end
```

distancepartition

```
function [partition] = distancepartition(V,E,initial)

% Distance Partition
%
% Input variables:
%
%   V:          Number of vertices of the graph G (natural number)
%   E:          Edge set of the graph G (n x 2-matrix)
%   initial:    Initial vertex for the distance partition.
%
% Output variables:
%
%   partition:      The distance partition obtained from the algorithm
%   edgesbetween:   The edges between the cells as a matrix
%
% This function is supposed to determine a distance partition for a given
% graph G(V,E), returned as an m x n-matrix, where n is the number of cells
% (equal to the diameter of the graph G), and m is the maximal number of
% vertices inside a cell. The column number corresponds with the n-1-th
% distance cell. If there are any zero's in the matrix, it means there are
```

```
% no more vertices inside the corresponding cell.

partition = initial;

% The first entry in the distance partition, is the initial vertex. From
% there, we are going to construct the further partition.

cellnumber = 1;
numberofedges = size(E,1);

% The algorithm has to check for each edge:
%   - Is one of the vertices contained in the previous cell?
%   - Is the other one not already contained in another cell?
%
% If these are both true, add this vertex to the current cell.

while nnz(partition) ~= V
cellnumber = cellnumber + 1;
vertexincell = 1;

    % As long as not all vertices (number of nonzero elements) have been
    % used for the partition, this loop has to go on. The next cell is
    % chosen (since for the last step, all adjacent vertices have been
    % determined) and the vertex number inside the cell is set again to
    % one.

for vertex = 1:V

    % Take a look at all different vertices
    for edge = 1:numberofedges

        % Take for each vertex a look at all edges, and determine whether
        % it is adjacent to the given vertex.

        if E(edge,1) == vertex
            previouscell = find(partition(:,cellnumber-1) == vertex, 1);

            % If so, look whether this vertex is contained in the previous
            % cell.

            if isempty(previouscell) == 0
                entirepartition = find(partition == E(edge,2), 1);

                % Look whether the other half is not already contained in
                % another partition (may be the previous cell, may be
                % another one).

                if isempty(entirepartition) == 1
                    partition(vertexincell,cellnumber) = E(edge,2);
                    vertexincell = vertexincell + 1;
```

```
                end

                % If the last is not true, we are going to put the outer
                % vertex in the current cell of the partition.

            end
        end

        if E(edge,2) == vertex

            % Same for the opposite version of the edges ([1 2] is the same
            % as [2 1], but both cases have to be treated).

            previouscell = find(partition(:,cellnumber-1) == vertex, 1);
            if isempty(previouscell) == 0
                entirepartition = find(partition == E(edge,1), 1);
                if isempty(entirepartition) == 1
                    partition(vertexincell,cellnumber) = E(edge,1);
                    vertexincell = vertexincell + 1;
                end
            end
        end
    end
end

end

end
```

distancepartitionnew

```
function [partition] = distancepartitionnew(adjacency_matrix, initial)

% Distance Partition (New)
%
% Input variables:
%
%   adjacency_matrix:   The adjacency matrix of the graph for which the
%                       distance partition has to be determined.
%   initial:            The initial vertex of the distance partition
%
% Output variables:
%
%   partition:          The distance partition obtained from the algorithm
%
% This function is supposed to determine a distance partition for a given
% graph G(V,E), returned as an m x n-matrix, where n is the number of cells
```

```
% (equal to the diameter of the graph G), and m is the maximal number of
% vertices inside a cell. The column number corresponds with the n-1-th
% distance cell. If there are any zero's in the matrix, it means there are
% no more vertices inside the corresponding cell.

A = adjacency_matrix;
partition(1,1) = initial;
V = size(A,1);

% The first entry in the distance partition is the initial vertex. From
% there, we are going to construct the remainder of the partition.

cellnumber = 1;

% The algorithm has to check for each edge:
%   - Is one of the vertices contained in the previous cell?
%   - Is the other one not already contained in another cell?
%
% If these are both true, add this vertex to the current cell.

while nnz(partition) ~= V

cellnumber = cellnumber + 1;

    % As long as not all vertices (number of nonzero elements) have been
    % used for the partition, this loop has to go on. The next cell is
    % chosen (since for the last step, all adjacent vertices have been
    % determined) and the vertex number inside the cell is set again to
    % one.

distancematrix = A^(cellnumber-1);
adjacencyrow = distancematrix(initial,:);
numberincell = 0;
verticesincell = 0;

    % The n-th power of the adjacency matrix is computed here. The (i,j)-th
    % element tells in how many ways vertex j can be reached from vertex i
    % in n steps. Since we are only interested in whether it is possible or
    % not, we have to decide which numbers are unequal to zero.

for vertex = 1:V

    % Look for each vertex whether it has distance n to the initial vertex.
    % Then check wheter this vertex is not already contained in the
    % partition. If both are true, put this vertex on the list
    % "verticesincell"

    if adjacencyrow(vertex) > 0
        inpartition = find(partition == vertex, 1);
        if isempty(inpartition) == 1
```

```
            numberincell = numberincell + 1;
            verticesincell(numberincell) = vertex;
        end
    end
end

% Now we have to put these vertices in the right order. We pick the
% one that is adjacent to the first vertex in the previous cell, and put it
% on top.

sourcevertex = partition(1,cellnumber-1);
adjacencyprevious = A(sourcevertex,verticesincell);

cell = zeros(numberincell,1); % Preallocation

for vertexincell = 1:numberincell
    if adjacencyprevious(vertexincell) == 1
        cell(1) = verticesincell(vertexincell);
        break
    end
end

sourcevertex = cell(1);
previouscell = partition(:,cellnumber-1);
previouscellcount = nnz(previouscell);

% Then, we look at the vertices that are adjacent to this one, inside
% the cell. If there aren't any, go to the next cell
for sourcevertexcount = 2:numberincell

adjacencyincell = A(sourcevertex,verticesincell);

% The vector "adjacencyincell" shows which vertices are adjacent to the
% source vertex.

if nnz(adjacencyincell) ~= 0

    % If this number is not equal to zero

    for previousvertex = 1:previouscellcount
    % We do not only have to look at the adjacency inside this cell, but
    % also to the connection with the previous cell. It is better that the
    % currently chosen vertices are adjacent to the first vertices in the
    % previous cell, since that will generate a better result for some
    % algorithms.

    for vertexincell = 1:numberincell
        if adjacencyincell(vertexincell) == 1
            currentvertex = verticesincell(vertexincell);
            checkvertex = previouscell(previousvertex);
```

```
            % The vertex that has to be checked, is the one that is
            % contained in the previous cell

            incell = find(cell == currentvertex, 1);
            checkprevious = A(currentvertex,checkvertex);
            if isempty(incell) == 1 && checkprevious == 1
                cell(sourcevertexcount) = currentvertex;
                sourcevertex = currentvertex;

                % From this point, we have to look which vertices are
                % adjacent to the recently chosen vertex. Therefore, this
                % one becomes the sourcevertex

                break
            end

            % When this point is reached, no vertex can be attached, so we
            % take the first vertex in row.

            for vertexindex = 1:numberincell
                incell = find(cell == verticesincell(vertexindex), 1);
                if isempty(incell) == 1
                    cell(sourcevertexcount) = verticesincell(vertexindex);
                    sourcevertex = cell(sourcevertexcount);
                    break
                end
            end
        end
    end

    if sourcevertex == currentvertex
        break
    end
    end
else

    % If the sourcevertex does not have any adjacent vertices in the
    % current cell, we just pick the first vertex that has not been chosen
    % yet.

    for vertexincell = 1:numberincell
        incell = find(cell == verticesincell(vertexincell), 1);
        if isempty(incell) == 1
            cell(sourcevertexcount) = verticesincell(vertexincell);
            sourcevertex = cell(sourcevertexcount);
            break
        end
    end
end
```

```
% Now, we can take a look at the next vertex in the cell, that already has
% been marked as "sourcevertex"

end % of "for sourcevertexcount = 2:numberincell"

% The entire cell has been created, so not it has to be added to the
% partition.

for vertexincell = 1:numberincell
    partition(vertexincell,cellnumber) = cell(vertexincell);
end

end % of "while nnz(partition) ~= V"

% By now, the complete partition must have been created

end
```

distancepartitionalg

```
function [numberofleaders] = distancepartitionalg(adjacency_matrix)

% DISTANCEPARTITIONALG
%
% Input Variables:
%   V: number of vertices of G (natural number)
%   E: edge set of G (n x 2-matrix)
%
% Output Variables:
%   leaderset: The leader set found accoring to the given algorithm
%   (V-dimensional row vector)

% Kanat's Algorithm for finding a zero forcing set for a given distance
% regular graph G(V,E):
%
% 1.    Determine the distance partition for a given vertex
% 2.    Assign the only vertex in the first cell to be a leader.
% 3.    Assume the n-th cell to be forced black, then look at how the
%       n+1-th cell can be forced black as well in the following way:
%       -   Compute the so-called "forward white degree" for each vertex in the
%           cell. If the property is, all these vertices have the same
%           forward degree (because of distance regularity). Name it c
%       -   Take c-1 vertices in the next cell as leaders, then run
%           zeroforcing
%       -   The forward white degrees have changed now, take the least one,
%           turn another c-1 into leaders.
```

```
%       -   Repeat this process, until the n+1-th cell has completely been
%           forced black.
% 4.    Inductively, the entire graph is forced black in this way.

A = adjacency_matrix;
V = size(A,1);
E = edgeset(A);

% Preallocation of variables:

leaderset = zeros(1,V);
whitevertices = ones(1,V);
blackvertices = zeros(1,V);

% 1. Determine the distance partition (since the graph is distance regular,
%    it does not matter which vertex we choose as out initial one):

partition = distancepartition(V,E,1);
diameter = size(partition,2) - 1;

% 2. Assign the only vertex in the first cell to be a leader (this
%    automatically becomes the first vertex):

leaderset(1) = 1;
blackvertices(1) = 1;

% 3. Induction step

for cell = 1 : diameter

    currentcell = partition(:,cell);
    nextcell = partition(:,cell + 1);

    currentvertices = nnz(currentcell);
    nextvertices = nnz(nextcell);

    % The number of nonzero elements in both cells, are equal to the number
    % of vertices inside the cells.

    % Determine the edges between the cells:
    edgesbetween = forwardedge(E,currentcell,nextcell);

    nextblack = 0;

    blackvertices = zeroforcing(V,blackvertices,E);
    whitevertices = ones(1,V) - blackvertices;

    for nextvertex = 1:nextvertices
        checkvertex = nextcell(nextvertex);
        if blackvertices(checkvertex) == 1
```

```
        nextblack = nextblack + 1;
    end
end

% Now we can start our loop for assigning new leaders

for i = 1:nextvertices

%%%%%%%%
%
% Bad idea to use this guy!:
%
% while nnz(nextblack) ~= nextvertices
%
%%%%%%%%%

% As long as not all vertices in the next cell are black, the following
% steps have to be repeated:

% Determine the white degree, for the first time
whitedegrees = forwardwhitedegree(V,whitevertices,edgesbetween);

minimalwhitedegree = min(whitedegrees);

if minimalwhitedegree == V
    break
end

% There must be only one, because of distance regularity.
minimalvertices = find(whitedegrees == minimalwhitedegree);
sourcevertex = minimalvertices(1);

% Since one of the minimal vertices must be chosen, it does not matter
% which one it is (this is not the case for the not-distance-regular
% graphs).

nextleaders = minimalwhitedegree - 1;

% Because n-1 leaders have to be assigned now

numberofedgesbetween = size(edgesbetween,1);
chosenleaders = 0;

for edge = 1: numberofedgesbetween

    if chosenleaders == nextleaders
        break
    end

    if edgesbetween(edge,1) == sourcevertex
```

```
            outervertex = edgesbetween(edge,2);
            if blackvertices(outervertex) == 0
                leaderset(outervertex) = 1;
                whitevertices(outervertex) = 0;
                blackvertices(outervertex) = 1;
                chosenleaders = chosenleaders + 1;
            end
        end

        % If the edge corresponds with the "source vertex", then assign the
        % outer edge to be a leader (only when the maximal number of
        % leaders has not yet been reached)


    end
    chosenleaders;
    % Now that the new leaders have been assigned, run zeroforcing on the
    % given edge set between the cells

    blackvertices = zeroforcing(V,blackvertices,E);
    whitevertices = ones(1,V) - blackvertices;

    % Now that new vertices have been forced black, check whether the
    % entire cell has already been forced black:

    nextblack = 0;
    % We check again from the beginning whether each next vertex is black
    % or not.

    for nextvertex = 1:nextvertices
        checkvertex = nextcell(nextvertex);
        if blackvertices(checkvertex) == 1
            nextblack = nextblack + 1;
        end
    end

    % If the number of black vertices is equal to the number of vertices in
    % the cell, then the entire cell has been forced black, and no other
    % leaders have to determined inside this cell.

    end

end

numberofleaders = nnz(leaderset);

leaderset;

end
```

distanceregular

```
function [V,E,partition] = distanceregular(F,B)

% DISTANCEREGULAR
%
% Input variables:
%
%   F:          Forward degrees in the intersection array (d-vector)
%   B:          Backward degrees in the intersection array (d-vector)
%
% Output variables:
%
%   V:          Number of vertices in the distance regular graph
%   E:          Edge set of the distance regular graph
%   partition:  Distance Partition of the given distance regular graph
%
%   The goal of this function, is to determine a distance regular graph (if
%   possible) for a certain given intersection array (and only given the
%   intersection array!). If it is not possible to produce such a graph, an
%   error is displayed.

if B(1) == 1
else
    disp('The first backward degree is not equal to one!')
    return
end

if numel(F) == numel(B)
else
    disp('The intersection array has not been built appropriately')
    return
end

diameter = numel(F)

vertices = zeros(1,diameter + 1)
partition = zeros(1,diameter + 1)
vertices(1) = 1
partition(1,1) = 1

valency = F(1)

for cell = 2:diameter + 1

    % - Determine the number of vertices in the next cell, if this isn't a
    %   natural number, abort the function
```

```
    % - Determine the number of edges between this cell and the previous
    %   one, and which vertices must be adjacent.
    % - Determine the edges inside the previous cell (which are not forward
    %   or backward connected).
    % - Go on until the entire partition has been created.

    k = vertices(cell-1)*B(



end

% Check whether the created graph is distance regular (according to the
% given properties)
end
```

drgalgs

```
function drgalgs(Adjacency_matrix)

vertices = size(Adjacency_matrix,1);
edges = edgeset(Adjacency_matrix);
numberofedges = size(edges,1);

leadersets = zeros(5,1)

disp('Cutting edge algorithm:')

leadersets(1) = cuttingedgealg(Adjacency_matrix);

disp('Distance partition algorithm:')

leadersets(2) = distancepartitionalg(Adjacency_matrix);

disp('The path algorithm')

leadersets(3) = pathalg(Adjacency_matrix);

disp('The path algorithm, revised version:')

leadersets(4) = pathalg2(Adjacency_matrix);

disp('The calculated maximum multiplicity of the adjacency matrix, which is also a lower b

drgmulti(vertices,edges);

leadersets
```

```
clear

end
```

drgmulti

```
function maxmultiplicity = drgmulti(V,E)

% DRGMULTI
%
% Input Variables:
%
% V:                The number of vertices in G (natural number).
% E:                The edge set of G (given as an n x 2-matrix).
%
% Output Variables:
%
% eigenvalues:      The found eigenvalues corresponding to the Laplacian
%                   Matrix of G.
% eigenvectors:     The found eigenvectors corresponding to the eigenvalues
%                   of the Laplacian of G.
%
% The purpose of this function, is to determine the eigenvalues and
% eigenvectors of the Laplacian of a distance regular graph G. These
% eigenvalues are equal to these of a tridiagonal matrix with the DRG
% parameters on the (sub)diagonals:
%
% First upper subdiagonal:  b's (forward degree)
% The diagonal itself:      a's (inner degree)
% First lower subdiagonal:  c's (backward degree)
%
% Then, the multiplicities of these eigenvalues are calculated, according
% to the formula for this multiplicity given in (Brouwer, Cohen, Neumaier;
% Distance Regular Graphs; Springer-Verlag, 1989), page 131.


partition = distancepartition(V,E,1);
diameter = size(partition,2) - 1;

% First, the function creates a distance partition, in order to calculate
% the forward, inner and backward degrees.

% Preallocation of variables:
a_s = zeros(diameter+1,1);
b_s = zeros(diameter,1);
c_s = zeros(diameter,1);
cellcount = zeros(diameter+1,1);

a_s(1) = 0;
```

```
cellcount(1) = 1;

% In the next while-loop, the DRG parameters are
% calculated, based on the
% distance partition of G:

for cell = 1:diameter
    current = partition(:,cell);
    next = partition(:,cell + 1);

    currentvertices = nnz(current);
    nextvertices = nnz(next);
    cellcount(cell+1) = nextvertices;

    edgesbetween = forwardedge(E,current,next);
    edgesinside =  forwardedge(E,next,next);

    b_s(cell) = size(edgesbetween,1)/currentvertices;
    c_s(cell) = size(edgesbetween,1)/nextvertices;

    a_s(cell + 1) = size(edgesinside,1)/nextvertices;
    % Even is there are no edges inside, this still corresponds to the
    % correct parameter a (which has to be zero in that case).

end

% If it is good, we have our integer parameters by now. Now we can
% construct the tridiagonal matrix L1:

L1 = diag(a_s,0) + diag(b_s,1) + diag(c_s,-1);

eigenvalues = eig(L1);

% Next, the so-called standard sequences of each of the eigenvalues have to
% be determined, based on the graph parameters and the eigenvalues:

valency = b_s(1);
standardsequences = zeros(diameter + 1,diameter + 1);

for eigenvalue = 1 : diameter + 1
    theta = eigenvalues(eigenvalue);
    standardsequences(1,eigenvalue) = 1;
    standardsequences(2,eigenvalue) = theta/valency;
    for cell = 3 : diameter + 1
        standardsequences(cell,eigenvalue) = ((theta - a_s(cell-1))/b_s(cell-1))*standards
    end
end

multiplicities = zeros(diameter + 1,1);
```

```
for eigenvalue = 1 : diameter + 1
    sequencevalues = zeros(diameter + 1,1);

    for cell = 1 : diameter + 1
        sequencevalues(cell) = cellcount(cell)*(standardsequences(cell,eigenvalue)^2);
    end

    multiplicities(eigenvalue) = V / sum(sequencevalues);

end

maxmultiplicity = max(multiplicities);

end
```

edgeset

```
function E = edgeset(adjacencymatrix)

% Input variables:
%
% adjacencymatrix:  The adjacency matrix of the given graph.
%
% Output variables:
%
% E:                The edge set which has to be determined.

% First, check whether the given adjacency matrix is appropriate, that is:
% - Square (#rows = #columns)
% - Logical (consists of only zeros and ones) [Logical(A) = A]
% - Symmetric (we're assuming the graph is undirected)
% - Not empty

A = adjacencymatrix;

if size(A,1) ~= size(A,2)
    disp('Inappropriate matrix sizes')
    return
end

if logical(A) ~= A
    disp('Matrix does not contain logical values')
    return
end

if isempty(nonzeros(A)) == 1
    disp('Adjacency matrix is empty')
    return
end
```

```
% Next, construct the edge set in the following way:
%
% - Take the first row of A, which corresponds to the first vertex
% - Look at each entry in the row:
%       If 1: create new edge
%       If 0: ignore
% - If the entire row has been inspected, take the next row, except for the
%   first entry (since A is symmetric)
% - Repeat this process, and ignore an additional entry when swapping to
%   the next row.

% Preallocation of variables:

% The number of edges is equal to half of the number of ones in the
% adjacency matrix, since it is symmetric. In fact, it is equal to half of
% the number of nonzero elements in the matrix:
numberofedges = size(nonzeros(A),1)/2;
E = zeros(numberofedges,2);

edgenumber = 1;
numberofvertices = size(A,1);

for row = 1:numberofvertices
    for column = row:numberofvertices
        if A(row,column) == 1
            E(edgenumber,1) = row;
            E(edgenumber,2) = column;
            edgenumber = edgenumber + 1;
        end
    end
end

end
```

forwardedge

```
function [betweenedges] = forwardedgenew(A,currentcell,nextcell)

% FORWARDEDGENEW
%
% Input variables:
%
%   A:              The adjacency matrix of the graph (n x 2-matrix)
%   currentcell:    The cell where the edges come from
%   nextcell:       The next cell to which the edges are connected
%
```

```
% Output variables:
%
%   betweenedges:   The edges that connect the two given cells.
%
%   This function determines the connections between two cells in a
%   distance partition, and displays them as an n x 2-matrix. The first
%   number in a row corresponds with a vertex inside the current cell, the
%   last one with the next cell.

currentvertices = nnz(currentcell);
nextvertices = nnz(nextcell);
betweennumber = 1;
betweenedges = [];

for currentvertexnumber = 1:currentvertices

    % For all the vertices in the current cell, the outgoing edges have to
    % be determined

    currentvertex = currentcell(currentvertexnumber);

    for nextvertexnumber = 1:nextvertices
        nextvertex = nextcell(nextvertexnumber);

        if A(currentvertex,nextvertex) == 1

            betweenedges(betweennumber,1) = currentvertex;
            betweenedges(betweennumber,2) = nextvertex;
            betweennumber = betweennumber + 1;
        end
    end
end
end
```

forwardwhitedegree

```
function [fwdwhitedegree] = forwardwhitedegree(V,whitevertices,E)

% Input Variables:
%
%   V:              number of vertices of G (natural number)
%   whitevertices:  the set of white vertices (row vector):
%                   - i'th element is 1 if vertex i is white
%                   - i'th element is 0 if vertex i is black
%   E:              edge set of G (n x 2-matrix)
%
% Output Variables:
%
%   fwdwhitedegree: the degrees of corresponding vertices (row vector)
```

```
%
% The objective of this function, is to determine the number of adjacent
% white vertices for each given (white) vertex in a directed (sub)graph,
% the so called "forward white degree".

% For some purposes, the minimal FWD has to be known, though this must be
% at least equal to 1. If a vertex has no white neighbours (perhaps due to
% the fact that only a part of the complete edge set is involved), the
% forward white degree is set equal to the number of vertices. In this way,
% the goal is still reached.

fwdwhitedegree = zeros(1,V);
numberofedges = size(E,1);
whiteneighbours = 0;

% Preallocation of variables

% Determine the white degree for each white vertex:

for vertex = 1:V
        for edge = 1:numberofedges

            if E(edge,1) == vertex && whitevertices(E(edge,2)) == 1
                whiteneighbours = whiteneighbours + 1;
            end
        end

        % Look at all the edges connected to the given vertex, and
        % determine whether they are white. If they are, raise up the
        % number of white neighbours.

        if whiteneighbours == 0
            fwdwhitedegree(vertex) = V;

            % Since the given white degrees must be unequal to zero, at
            % least the number V is an upper bound for the minimal white
            % degree.

        else
            fwdwhitedegree(vertex) = whiteneighbours;
        end

        whiteneighbours = 0;
end

end
```

pathalg

```
function [numberofleaders] = pathalg(adjacency_matrix)

% PATHALG
%
% Input Variables:
%   V: number of vertices of G (natural number)
%   E: edge set of G (n x 2-matrix)
%
% Output Variables:
%   leaderset: The leader set found accoring to the given algorithm
%   (V-dimensional row vector)

% The found path algorithm for creating zero forcing sets for a given
% (distance regular) graph:
%
% 1.    Determine the distance partition for a given vertex
% 2.    Assign the only vertex in the first cell to be a leader.
% 3.    Create a path of black vertices from one side of the distance
%       partition to another, in the following way:
%       -   Start in the initial vertex (= initial cell)
%       -   Take at random a vertex from the next cell, assign it as a
%           leader, and look whether other vertices are forced to black.
%       -   If there is a black vertex in the next cell, jump to the cell
%           after that one, and repeat the process.
%       -   Go on this way, until a path of black vertices is created from
%           the other side of the distance partition to the other.
% 4.    Create new paths paths from black forced cells, recursively, in the
%       following way:
%       -   Start in the last cell that is entirely forced to black.
%       -   Take the first vertex in the cell (from above) that has no
%           black neighbours in the next cell.
%       -   Repeat the same process as in the previous step, until a path
%           of black vertices from one side to the other is reached.
% 5.    Keep creating paths, until the entire vertex set is forced black.

% Preallocation of variables:
A = adjacency_matrix;
V = size(A,1);
E = edgeset(A);

leaderset = zeros(1,V);
whitevertices = ones(1,V);
blackvertices = zeros(1,V);

% 1. Determine the distance partition (since the graph is distance regular,
%    it does not matter which vertex we choose as out initial one):
```

```
partition = distancepartition(V,E,1);
diameter = size(partition,2) - 1;

% 2. Assign the only vertex in the first cell to be a leader (this
%    automatically becomes the first vertex):

leaderset(1) = 1;
blackvertices(1) = 1;

% 3. Create a path of black vertices from one side of the distance
%    partition to another.

sourcevertex = 1;

for cell = 1 : diameter

    currentcell = partition(:,cell);
    nextcell = partition(:,cell + 1);

    currentvertices = nnz(currentcell);
    nextvertices = nnz(nextcell);

    % Look which vertices are adjacent to the source vertex
    adjacentedges = forwardedge(E,sourcevertex,nextcell);
    adjacentvertices = adjacentedges(:,2);
    numberofadjacent = nnz(adjacentvertices);

    for vertex = 1 : numberofadjacent
        testvertex = adjacentvertices(vertex);

        % Check whether this vertex is still black. If it isn't, assign
        % this one as a leader, take it as a sourcevertex, run zeroforcing
        % and go on to the next cell.

        if whitevertices(testvertex) == 1
            leaderset(testvertex) = 1
            blackvertices(testvertex) = 1
            whitevertices(testvertex) = 0
            sourcevertex = testvertex;
            break
        end

        if vertex == numberofadjacent

        % If the script has reached this point, it means there are no black
        % vertices adjacent to this one, so we can jump to the next cell

            sourcevertex = adjacentvertices(1);
        end
    end
```

```
    blackvertices = zeroforcing(V,leaderset,E);
    whitevertices = ones(1,V) - blackvertices;

end

% At this point, the first path is created. From now on, we start to create
% other paths, starting at vertices that have not yet been forced to zero

while nnz(blackvertices) ~= V

% As long as not all vertices are black, the process has to continue.

% First, determine which cell is the last one which is entirely black:

for cell = 1:diameter

    currentcell = partition(:,cell);
    nextcell = partition(:,cell + 1);

    currentvertices = nnz(currentcell);
    nextvertices = nnz(nextcell);

    blackincell = blackvertices(nextcell(1:nextvertices));

    if nnz(blackincell) == nextvertices
    else
        % If the next cell is not entirely black, the current cell has to
        % be the source cell in order to create additional paths.

        sourcecell = currentcell;
        sourcecellnumber = cell;
        break
    end
end

% The correct cell is now called sourcecell, so we determine which next
% path of leaders (or otherwise black vertices) we can create.

sourcevertices = nnz(sourcecell);

for vertexnumber = 1 : sourcevertices

    sourcevertex = sourcecell(vertexnumber);

    for cell = sourcecellnumber : diameter

    currentcell = partition(:,cell);
    nextcell = partition(:,cell + 1);
```

```
    currentvertices = nnz(currentcell);
    nextvertices = nnz(nextcell);

    % Look which vertices are adjacent to the source vertex
    adjacentedges = forwardedge(E,sourcevertex,nextcell);
    adjacentvertices = adjacentedges(:,2);
    numberofadjacent = nnz(adjacentvertices);

    for vertex = 1 : numberofadjacent
        testvertex = adjacentvertices(vertex);

        % Check whether this vertex is still black. If it isn't, assign
        % this one as a leader, take it as a sourcevertex, run zeroforcing
        % and go on to the next cell.

        if whitevertices(testvertex) == 1
            leaderset(testvertex) = 1
            blackvertices(testvertex) = 1;
            whitevertices(testvertex) = 0;
            sourcevertex = testvertex;
            break
        end

        if vertex == numberofadjacent

        % If the script has reached this point, it means there are no black
        % vertices adjacent to this one, so we can jump to the next cell

            sourcevertex = adjacentvertices(1);
        end
    end

    blackvertices = zeroforcing(V,leaderset,E)
    whitevertices = ones(1,V) - blackvertices

    end

    if nnz(blackvertices) == V
        break
    end
end

end % Corresponding to while-loop

numberofleaders = nnz(leaderset);

leaderset;
```

```
end
```

pathalg2

```
function [numberofleaders] = pathalg2(adjacency_matrix)

% PATHALG2
%
% Input Variables:
%   V: number of vertices of G (natural number)
%   E: edge set of G (n x 2-matrix)
%
% Output Variables:
%   leaderset: The leader set found accoring to the given algorithm
%   (V-dimensional row vector)

% The found path algorithm for creating zero forcing sets for a given
% (distance regular) graph:
%
% 1.    Determine the distance partition for a given vertex
% 2.    Assign the only vertex in the first cell to be a leader.
% 3.    Create a path of black vertices from one side of the distance
%       partition to another, in the following way:
%       -   Start in the initial vertex (= initial cell)
%       -   Take at random a vertex from the next cell, assign it as a
%           leader, and look whether other vertices are forced to black.
%       -   If there is a black vertex in the next cell, jump to the cell
%           after that one, and repeat the process.
%       -   Go on this way, until a path of black vertices is created from
%           the other side of the distance partition to the other.
% 4.    Create new paths paths from black forced cells, recursively, in the
%       following way:
%       -   Start in the last cell that is entirely forced to black.
%       -   Take the first vertex in the cell (from above) that has no
%           black neighbours in the next cell.
%       -   Repeat the same process as in the previous step, until a path
%           of black vertices from one side to the other is reached.
% 5.    Keep creating paths, until the entire vertex set is forced black.

% Preallocation of variables:
A = adjacency_matrix;
V = size(A,1);
E = edgeset(A);

leaderset = zeros(1,V);
whitevertices = ones(1,V);
blackvertices = zeros(1,V);
```

```
% 1. Determine the distance partition (since the graph is distance regular,
%    it does not matter which vertex we choose as out initial one):

partition = distancepartitionnew(A,1);
diameter = size(partition,2) - 1;

% 2. Assign the only vertex in the first cell to be a leader (this
%    automatically becomes the first vertex):

leaderset(1) = 1;
blackvertices(1) = 1;

% 3. Create a path of black vertices from one side of the distance
%    partition to another.

sourcevertex = 1;

for cell = 1 : diameter

    currentcell = partition(:,cell);
    nextcell = partition(:,cell + 1);

    currentvertices = nnz(currentcell);
    nextvertices = nnz(nextcell);

    % Look which vertices are adjacent to the source vertex
    adjacentedges = forwardedge(E,sourcevertex,nextcell);
    adjacentvertices = adjacentedges(:,2);
    numberofadjacent = nnz(adjacentvertices);

    for vertex = 1 : numberofadjacent
        testvertex = adjacentvertices(vertex);

        % Check whether this vertex is still black. If it isn't, assign
        % this one as a leader, take it as a sourcevertex, run zeroforcing
        % and go on to the next cell.

        if whitevertices(testvertex) == 1
            leaderset(testvertex) = 1;
            blackvertices(testvertex) = 1;
            whitevertices(testvertex) = 0;
            sourcevertex = testvertex;
            break
        end

        if vertex == numberofadjacent

        % If the script has reached this point, it means there are no black
        % vertices adjacent to this one, so we can jump to the next cell
```

```
            sourcevertex = adjacentvertices(1);
        end
    end

    blackvertices = zeroforcing(V,leaderset,E);
    whitevertices = ones(1,V) - blackvertices;

end

% At this point, the first path is created. From now on, we start to create
% other paths, starting at vertices that have not yet been forced to zero

while nnz(blackvertices) ~= V

% As long as not all vertices are black, the process has to continue.

% First, determine which cell is the last one which is entirely black:

for cell = 1:diameter

    currentcell = partition(:,cell);
    nextcell = partition(:,cell + 1);

    currentvertices = nnz(currentcell);
    nextvertices = nnz(nextcell);

    blackincell = blackvertices(nextcell(1:nextvertices));

    if nnz(blackincell) == nextvertices
    else
        % If the next cell is not entirely black, the current cell has to
        % be the source cell in order to create additional paths.

        sourcecell = currentcell;
        sourcecellnumber = cell;
        break
    end
end

% The correct cell is now called sourcecell, so we determine which next
% path of leaders (or otherwise black vertices) we can create.

sourcevertices = nnz(sourcecell);

for vertexnumber = 1 : sourcevertices

    sourcevertex = sourcecell(vertexnumber);

    for cell = sourcecellnumber : diameter
```

```
    currentcell = partition(:,cell);
    nextcell = partition(:,cell + 1);

    currentvertices = nnz(currentcell);
    nextvertices = nnz(nextcell);

    % Look which vertices are adjacent to the source vertex
    adjacentedges = forwardedgenew(A,sourcevertex,nextcell);
    adjacentvertices = adjacentedges(:,2);
    numberofadjacent = nnz(adjacentvertices);

    for vertex = 1 : numberofadjacent
        testvertex = adjacentvertices(vertex);

        % Check whether this vertex is still black. If it isn't, assign
        % this one as a leader, take it as a sourcevertex, run zeroforcing
        % and go on to the next cell.

        if whitevertices(testvertex) == 1
            leaderset(testvertex) = 1;
            blackvertices(testvertex) = 1;
            whitevertices(testvertex) = 0;
            sourcevertex = testvertex;
            break
        end

        if vertex == numberofadjacent

        % If the script has reached this point, it means there are no black
        % vertices adjacent to this one, so we can jump to the next cell

            sourcevertex = adjacentvertices(1);
        end
    end

    blackvertices = zeroforcing(V,leaderset,E);
    whitevertices = ones(1,V) - blackvertices;

    end

    if nnz(blackvertices) == V
        break
    end
end

end % Corresponding to while-loop

numberofleaders = nnz(leaderset);
```

```
leaderset;


end
```

whitedegree

```
function [whitedegree] = whitedegree(V,whitevertices,E)

% Input Variables:
%
%   V:              number of vertices of G (natural number)
%   whitevertices:  the set of white vertices (row vector):
%                   - i'th element is 1 if vertex i is white
%                   - i'th element is 0 if vertex i is black
%   E:              edge set of G (n x 2-matrix)
%
% Output Variables:
%
%   whitedegree:    the degrees of corresponding vertices (row vector)
%
% The objective of this function, is to determine the number of adjacent
% white vertices for each given (white) vertex, the so-called "white
% degree".

whitedegree = zeros(1,V);
numberofedges = size(E,1);
whiteneighbours = 0;

% Preallocation of variables

% Determine the white degree for each white vertex:

for vertex = 1:V
    if whitevertices(vertex) == 1

        % We only have to look at vertices that are still white, since we
        % have to choose some of them to be leaders.

        for edge = 1:numberofedges
            if E(edge,1) == vertex && whitevertices(E(edge,2)) == 1
                whiteneighbours = whiteneighbours + 1;
            elseif E(edge,2) == vertex && whitevertices(E(edge,1)) == 1
                whiteneighbours = whiteneighbours + 1;
            end
        end
```

```
        % Look at all the edges connected to the given vertex, and
        % determine whether they are white. If they are, raise up the
        % number of white neighbours.

        whitedegree(vertex) = whiteneighbours;

        whiteneighbours = 0;
    else
        whitedegree(vertex) = V;

        % For each given black vertex, we need an upper bound as well,
        % which is useful for the next step (determining the minimal white
        % degree).
    end
end

end
```

zeroforcing

```
function [blackset] = zeroforcing(numberofvertices,leaderset,edgeset)

% ZEROFORCING (Undirected)
%
% Input Variables:
%
%   numberofvertices:          number of vertices of G (natural number)
%   leaderset:          the set of leader vertices (row vector):
%               - i'th element is 1 if vertex i is white
%               - i'th element is 0 if vertex i is black
%   edgeset:           edge set of G (n x 2-matrix)
%
% Output Variables:
%
%   blackset:   the vertex set which is forced to zero by the given leader
%               set (row vector).
%
% The objective of this function, is to determine which vertices can be
% forced to zero, out of given vertex, edge and leader sets for a graph
% G(V,E). In the following function, a loop will determine the possible
% white neighbours for each leader vertex, and when there is exactly one,
% this one will be turned into a black one. This process wiil go on until
% the black set does not change anymore.

% Preallocation of variables:

newblackset = leaderset;
oldblackset = zeros(1,numberofvertices);
```

```
% Unless L = V (which is clearly a zero forcing set), at least one step has
% to be taken

numberofedges = size(edgeset,1);
whiteneighbours = 0;
numberwhite = 0;

for step = 1:numberofvertices

if newblackset == oldblackset
else

    % This loop goes on until the black set does not change anymore.
    oldblackset = newblackset;
    blackset = oldblackset;

    for vertex = 1:numberofvertices

    % Look whether you got a black vertex or a white one.

        if blackset(vertex) == 1

        % Take the entire edge set, and look at all possible connections.
        % Look whether the vertices are adjacent to the chosen leader
        % vertex in the loop (only the possibility [vertex, neighbour]!).

            for edge = 1:numberofedges
                if edgeset(edge,1) == vertex
                    neighbour = edgeset(edge,2);

                % Look whether the adjacent vertex is a white
                % neighbour:
                    % True: add to the list "whiteneighbours"
                    % False: ignore the adjacent vertex

                    if blackset(neighbour) == 0
                        numberwhite = numberwhite + 1;
                        whiteneighbours(numberwhite) = neighbour;
                    end
                elseif edgeset(edge,2) == vertex

                % Do the same, but then for the possibility [neighbour,
                % vertex].

                    neighbour = edgeset(edge,1);
                    if blackset(neighbour) == 0
                        numberwhite = numberwhite + 1;
                        whiteneighbours(numberwhite) = neighbour;
                    end
                end
```

```
            end
            numberwhite = 0;

            % Look whether there is exactly one white neighbour for the
            % chosen leader vertex, and the value of whiteneighbours it not
            % equal to zero (the pre-assigned value).
                % True, turn the adjacent white vertex into a black one.
                % False, ignore the vertex
            if size(whiteneighbours,2) == 1 && whiteneighbours(1) ~= 0
                blackset(whiteneighbours(1)) = 1;
            end

            whiteneighbours = 0;
            newblackset = blackset;

            % Clear the list of white neighbours, since we take a look at
            % the next vertex from now on (if it exists).
        end

        %If the vertex is white, here we go on.
    end

    %If we have determined a possible zero forcing step for each vertex,
    %then here we are again. This loop goes on until the black set does not
    %change anymore.
end

end

blackset = newblackset;

% Even if no steps are done, the maximal set of vertices forced to zero is
% equal to the given leader set, which is the initial value of
% "oldblackset".
end
```

zeroforcingpossibilities

```
V = input('How many vertices does the graph contain? \n \n');

E = input('\n Give up the edge set as an n x 2-matrix: \n \n Left: one end of the edge \n

L = zeros(2^V,V);

% Preallocation of L

% At this point, all possible leader sets have to be determined. Since
```

```
% there are n vertices, there are 2^n possible leader sets. All of them are
% contained within a truth table:

for i = 1:2^V
    L(i,:) = de2bi(i-1,V,'left-msb');
end

% With this for loop, a complete truth table is generated, where each row
% stand for all possible leader sets, and the i-th element of each row
% corresponds to the i-th vertex.

forcing = 0;
setnumber = 0;

for i = 1:2^V
    blackset = zeroforcing(V,L(i,:),E);

    % Check for each possible set whether it is a zero forcing set.

    if blackset == ones(1,V)
        setnumber = setnumber + 1;
        forcingsets(setnumber,:) = L(i,:);
        forcingnumbers(setnumber,1) = norm(L(i,:),1);
    end

    % When this set is indeed a leader set, put it in the matrix
    % "forcingsets", and the corresponding number of leaders on the list
    % "forcingnumbers"

end

zeroforcingnumber = min(forcingnumbers);
minimalindex = 0;

for i = 1 : size(forcingsets,1)
    if forcingnumbers(i) == zeroforcingnumber
        minimalindex = minimalindex + 1;
        minimalsets(minimalindex,:) = forcingsets(i,:);
    end
end

% Check whether a leader set is minimal, and take all of the minimal sets

minimalsets
zeroforcingnumber
```