KTH Computer Science
and Communication

The Monk Problem

Verifier, heuristics and graph decompositions for a pursuit-evasion problem with a node-located evader

FREDRIKSSON, BASTIAN
bastianf@kth.se

LUNDBERG, EDVIN
edvinlun@kth.se

Bachelor's thesis, KTH CSC
Supervisor: Arvind Kumar
Examiner: Örjan Ekeberg

May 10, 2015

Abstract

This paper concerns a specific pursuit-evasion problem with a node-located evader which we call *the monk problem*. First, we propose a way of verifying a strategy using a new kind of recursive systems, called EL-systems. We show how an EL-system representing a graph-instance of the problem can be represented using matrices, and we give an example of how this can be used to efficiently implement a verifier.

In the later parts we propose heuristics to construct a strategy, based on a greedy algorithm. Our main focus is to minimise the number of pursuers needed, called *the search number*. The heuristics rely on properties of minimal stable components.

We show that the minimal stable components are equivalent to the strongly connected components of a graph, and prove that the search number is equal to the maximum search number of its strongly connected components. We also establish lower and upper bounds for the search number to narrow the search space.

Referat

Denna rapport avhandlar ett specifikt pursuit-evasion problem med en hörnplacerad flykting, som vi kallar för *munkproblemet*. Först föreslår vi ett sätt att verifiera en strategi med en ny typ av rekursivt system, kallat EL-system. Vi visar hur ett EL-system som representerar en grafinstans av munkproblemet kan representeras med matriser, och vi ger ett exempel på hur detta kan användas för att effektivt implementera en verifikator.

I de senare delarna föreslår vi heuristiker för att konstruera en strategi, baserad på giriga algoritmer. Vårt huvudfokus är att minimera antalet förföljare som krävs för att dekontaminera grafen, det så kallade *söktalet*. Vår heuristik förlitar sig på egenskaper för minimala stabila komponenter.

Vi visar att minimala stabila komponenter är ekvivalenta med de starka komponenterna i en graf, och härleder att söktalet är lika med det maximala söktalet för grafens starka komponenter. Vi etablerar också undre och övre gränser för söktalet i syfte att minska sökintervallet.

Contents

Chapter 1

Background

Pursuit-evasion problems are a family of problems (see figure 1.1) where the goal is to find one or more evaders moving in an environment. The most generic description of the problem is as follows: By using one or more pursuers, how can you guarantee the capture of all evaders, or as many evaders as possible; in the shortest amount of time, or with the smallest amount of resources and how many pursuers are needed? The problem is considered completely solved when all, or as many evaders as possible, are captured by the pursuers. Thus the solution describes how many pursuers are needed (also known as *the search number*), and how the pursuers must move in order to guarantee capture of all evaders (also known as *a strategy*).

1.1 Related work

Pursuit-evasion problems was initially seen as *search games*. The first search game which is built upon a pursuit-evasion problem was invented in 1965 by Rufus Isaacs, when he described the so called princess-monster game. The Princess-Monster game is an example of a differential game. Differential games are continuous versions of the pursuit-evasion problem and the movement of evaders and pursuers are modeled using differential equations. In The Princess-Monster game, a princess tries to avoid a monster in a dark room of arbitrary shape. The princess is captured when she comes within a certain distance of the monster. The monster and the princess are moving along known trajectories (paths in space). The monster is moving with constant speed while the princess can move arbitrarily fast [8].

Other search games include the well-known Cops and Robbers [1], Helicopter Cops and Robbers [14] and The Hunter-Rabbit game [9]. They are all discrete versions of the pursuit-evasion problem. Such problems can be modeled with a graph consisting of vertices and edges.

In Cops and Robbers, there are cops tracking down one or more robbers. The cops and the robbers are moving in turns. At each turn, one can either remove a cop, place a cop or move a cop one step (slide a cop along an edge). Helicopter Cops and Robbers is very similar to Cops and Robbers, but cops are allowed to jump to

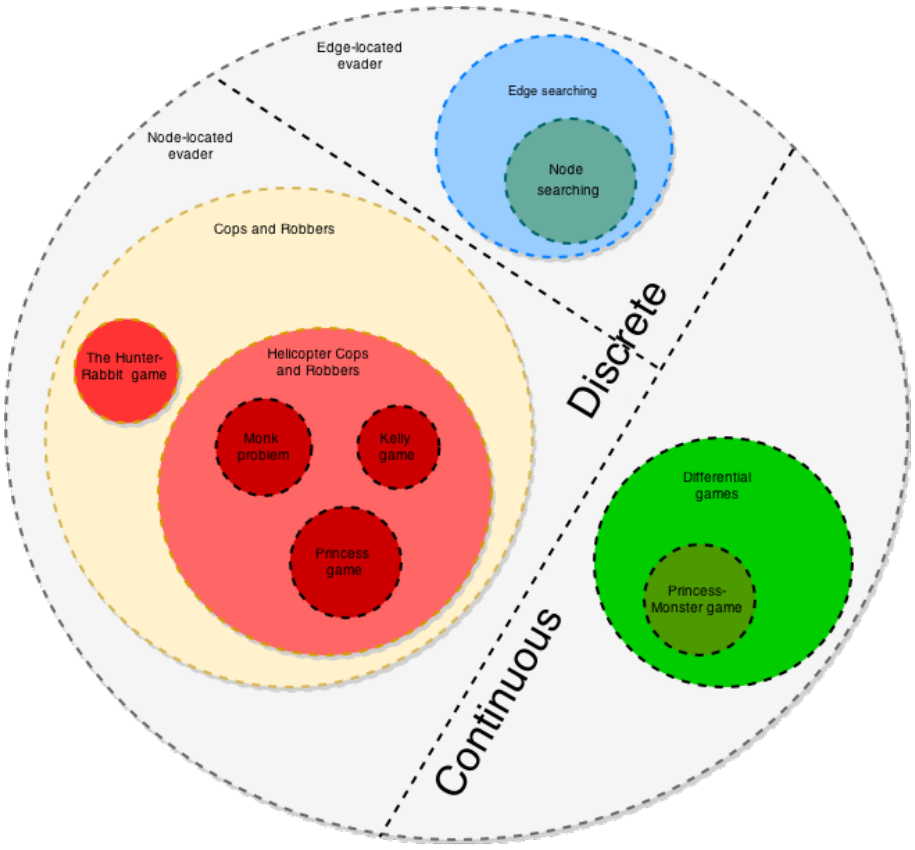

Figure 1.1. Classes of pursuit-evasion problems.

any vertex in the graph in one step. The Kelly game is very similar to Helicopter Cops and Robbers, but the robber is inert, meaning that it only moves when a cop is about to occupy his position. When this happens, the robber is allowed to move along any path in the graph not guarded by a cop [7]. Another variant is the Princess game, where a prince tries to find a princess in a palace modeled with a directed graph. The princess is forced to move between two adjacent rooms on each night. On each day, the prince announces in which room he want to look during the next day, whereby the princess moves in such a way that she will avoid the prince if she can. The princess is captured when she is forced to move into the room that the prince is due to visit. In The Hunter-Rabbit game, the goal is for a hunter to capture a rabbit. Both the rabbit and the hunter are aware of each other's positions. The hunter and the rabbit are taking turns in moving, and they are allowed to move one step at each turn [2].

Aigner et. al. showed that in Cops and Robbers, three cops are enough to capture all evaders in a planar graph [1]. Fomin et. al proved that the computation of the search number for Cops and Robbers is NP-hard on general graphs [6]. Paul Hunter and Stephan Kreutzer showed that the search number for the Kelly game

is equal to the minimal width of all its Kelly decompositions, called Kelly width, and that it is NP-hard to find this number. The Kelly width can be bounded from above by repeatedly removing vertices from the graph and adding edges to preserve reachabilility [7]. Furthermore, it is possible to determine if a graph has Kelly width at most two in polynomial time [12]. John Britnell and Mark Wildon proved that it is possible to find the princess in linear time for all palaces which do not contain a cycle or a subgraph isomorphic with a star consisting of three branches of length three [2].

What Cops and Robbers and similar games have in common are that the pursuers are looking for a node-located evader. That is, the evaders are hiding in the vertices of the graph, and an evader is usually captured when a pursuer occupies his position. Another variant is where the pursuers are looking for an *edge-located evader*. This problem is known as edge searching, which allows the same moves as in Cops and Robbers, but in edge searching, an evader is captured when a pursuer is moving along an edge. The most common variant of edge searching is *node searching*, which was initially stated by Lefteris Kirousis and Christos Papadimitriou in 1986. One might think of this as robots searching for terrorists in a cave. The terrorists are hiding in the tunnels of the cave and the vertices represent intersection points where two tunnels meet. An evader hiding in an edge $v_1 \to v_2$ is captured when there are pursuers positioned at both v_1 and v_2 at the same time [11]. This problem has been heavily studied in literature due to the fact that the problem is equivalent to many other important problems; such as interval thickness, the gate-matrix layout problem, and the narrowness problem. Finding the search number for an edge-located evader is known to be NP-complete, but the problem can be solved efficiently for some types of graphs, including trees, cographs, block graphs, permutation graphs, k-starlike graphs and partial k-trees ($k \geq 1$) [3].

1.2 L-system

The L-system or Lindenmayer-system was introduced by Aristid Lindenmayer in 1968. The L-system is a recursive system which is a type of formal grammar. The L-system consists of an alphabet of symbols, a starting state and production rules. A state consists of a set of symbols. A production rule will map a set of symbols to a symbol.

The L-system iteratively updates a state. Each state will get expanded by applying as many production rules as possible which will produce the next state [13]. The relevance of L-systems can be seen in section 3.1.

1.3 Applications

Pursuit-evasion problems have several practical applications. Differential games were used to design missile guidance systems [8]. A pursuit-evasion problem with a node-located evader, called Seepage, was used to model lava flow from the Eldfell volcano

on Iceland [4]. Heuristics for node searching is used to design VLSI-layouts [5]. Our problem could be used to describe strategies for combat- or search and rescue operations, decontaminating a computer network, or as a graph searching algorithm.

Chapter 2

Introduction

In this paper we will study a generalisation of The Princess game which we call *the monk problem*. While The Princess game is constrained to undirected graphs with one pursuer, the monk problem allows for an arbitrary number of pursuers in any directed graph meeting the criteria in 2.3. The name (and the rules) of the problem is inspired by the original problem presented to the authors by Dilian Gurov.

2.1 Original problem

The problem by Dilian Gurov can be explained as follows: There exists a monk in the mountains. In the mountains there are five caves lined up linearly. The monk sits in a cave during the day and meditates. Then every night he moves to an adjacent cave. If you are allowed to look in exactly one of the five caves every day; Can you guarantee that you find the monk and how many days are needed?

The exact origin of the problem is unknown to the authors.

2.2 Overview

The purpose of this paper is to study techniques for solving and verifying solutions to the monk problem for different graph structures.

In section 3.1 we construct a verifier, based on EL-systems. We prove that the verifier is correct and give examples of how it operates. In 3.1.1 we transform an EL-system into a timeline, which can be used to verify a solution by hand. In section 3.3 and 3.4 we establish the theoretical foundation for solving the problem. To do this, we construct two decomposition algorithms, called *stable component decomposition* and *cycle decomposition*. We prove that stable component decomposition partitions the graph in such a way that each part can be solved separately. We also show that cycle decomposition can be used to solve parts of a graph in linear time, as well as giving an upper bound for the number of pursuers. In section 3.6 we propose

heuristics for efficiently solving the monk problem with as few pursuers as possible, based on the theorems presented in earlier sections.

2.3 Problem definition

When referring to the monk problem in this paper, we use the following problem definition. Note that the evaders and pursuers reside in the vertices of a graph only, similar to Cops and Robbers.

Definition 1 (Monk graph)**.** *A monk graph is a finite graph G with the following properties:*

- *G is a directed graph without multiple edges*
- *G has at least one vertex*
- *G consists of exactly one component*
- *G is allowed to contain self-loops (a vertex having an edge to itself)*
- *It is possible to follow an edge from every vertex in the graph (no dead ends) if the graph is not a singleton.*

We allow a single vertex without edges to fall into the definition of a monk graph. Such a monk graph is called a singleton.

Definition 2 (The monk problem)**.** *Given a monk graph G, the monk problem consists of answering the following two questions:*

- *What is the search number (minimum number of pursuers) required to guarantee capture of all evaders?*
- *Given p pursuers, how should they move in order to find all evaders in the shortest amount of time?*

In this paper, we will mainly focus on approximating the search number.

The evaders and the pursuers take turns in moving. On each move (day), the pursuer might look at (decontaminate) any vertex v in the graph or stay idle. If one or more evaders reside in v, they are captured. An evader residing in the vertex v_1 is forced to move along exactly one edge $v_1 \to v_2$ on each turn.

Definition 3 (The monk search number decision problem)**.** *The monk search number decision problem (MSNDP) is to determine, given a monk graph G and an integer k, whether G can be decontaminated in a finite number of steps using k or less pursuers.*

Definition 4 (The monk strategy length decision problem)**.** *The monk strategy length decision problem (MSLDP) is to determine, given a monk graph G an integer k and an integer l whether there exists a winning strategy for G using k pursuers of length l or less.*

We will view the vertices in the graph as contaminated (uncaught evader can be there) and decontaminated (uncaught evader can not be there).

Definition 5 (Contamination)**.** *A vertex $v \in V(G)$ is contaminated on day n if, and only if, an uncaught evader can reside in v on day n. All vertices in G are contaminated on the initial day.*

Definition 6 (Decontamination)**.** *A vertex $v \in V(G)$ is decontaminated on day n if, and only if no uncaught evader can reside in v on day n, that is:*

- *Direct decontamination: a pursuer decontaminates v on day n where $n \geq 0$, or*
- *Indirect decontamination: all vertices in $\mathbb{V}(\mathbb{G})$ where $\mathbb{V}(\mathbb{G}) = \{u | u \to v \in E(G)\}$ are decontaminated on day $n-1$ where $n > 0$.*

Definition 7 (Recontamination)**.** *If $v \in V(G)$ is contaminated on day n, for all neighbours v' of v: v' will be contaminated on day $n+1$ unless a pursuer decontaminates v' on day $n+1$. A recontamination occurs when a decontaminated vertex becomes contaminated on the next day.*

With this notation, we can define search strategies.

Definition 8 (Pursuer strategy)**.** *A pursuer strategy describes the search order of a pursuer, that is, the vertex being decontaminated by the pursuer on each day. The pursuer strategy may also contain idle entries, that is a day where the pursuer is not decontaminating anything. The length of a pursuer strategy is the number of days in the pursuer strategy, including the days where the pursuer stays idle.*

Definition 9 (Strategy)**.** *A strategy with p pursuers consists of p pursuer strategies of length k. Such a strategy is called a winning strategy if the pursuers are able to decontaminate all vertices in a finite number of steps.*

A strategy is equivalent to a solution for the graph. Furthermore, we can define exactly what the search number is and what an optimal strategy is.

Definition 10 (Search number)**.** *The search number of a monk graph is the minimum amount of pursuers for which there exists a winning strategy.*

Definition 11 (Optimal strategy)**.** *An optimal strategy is a winning strategy with the search number, and with the smallest length of the strategy.*

When decomposing a graph into subgraphs, for example using stable component decomposition described in 3.3, some components might be singletons. Here, we define the optimal strategy for a singleton.

Definition 12 (Singleton strategy)**.** *The optimal strategy for a singleton is a strategy with zero pursuers staying idle for one day.*

2.4 Terminology

When talking about graphs, it is useful to have some terminology.

Given a graph $G = (V, E)$ describing an environment, we denote the vertex set of G with $V(G)$ and the edge set of G with $E(G)$.

A directed edge from u to v is denoted as $u \to v$. An undirected edge between u and v is denoted as $u \longleftrightarrow v$.

The number of vertices in a graph, or the order of a graph, is written as $|V(G)|$, $|V|$ or $|G|$. The number of edges in a graph is written as $|E|$ or $|E(G)|$.

2.5 Source code

Test cases and implementations of the algorithms described in this paper are available in our GitHub repository: `https://github.com/Realiserad/kex15`

Chapter 3

Results

3.1 EL-system

The EL-system is an extension of the L-System which is an example of a *recursive system*. The EL-system utilises one or more stacks representing the strategy for the specific instance of the monk problem. Each stack corresponds to a pursuer strategy. We use the EL-system to describe the set of decontaminated vertices on each day.

The EL-system could be used for other applications, so the following is the general definition.

Definition 13 (The EL-system)**.** *An EL-system is a tuple* $\phi = (A, \omega, P, \sigma)$ *where:*

- A *is the alphabet of symbols*
- ω *is the starting set of symbols, called the initial state*
- P *is a set of production rules. A production rule is seen as a function* $p_B : \mathcal{P}(A) \to A$ *where* $\mathcal{P}(A)$ *is the power set of* A*.* $p_B(A') = c$*,* $c \neq \varnothing$ *if, and only if,* $B \subseteq A'$*. Intuitively if* $p_B(A') = c$*, then* A' *contains the symbols which produce* c*. When* p_B *is applied on a state* $A' \subseteq A$*, the value returned is* $p_B(A')$*. We denote* c *as the produced value of* p_B*.*
- σ *is a set of stacks where each element* s *in a stack:* $s \in A$

The EL-system iteratively updates a working state. We formally define this process as follows.

Definition 14 (EL-system production process (ELPP))**.** *The EL-system production process is an algorithm that works as follows:*

1. *Let* $S_0 = \omega$*.*
2. *If the stacks are empty then STOP.*
3. *For each stack, pop the next element* E *from the stack and let* $S_i := S_i \cup \{E\}$*.*

4. *Parse S_i and apply as many production rules as possible. Assign the result to S_{i+1}.*

5. *Goto 2.*

The result state can then be found in S_k where k is the number of iterations passed until all stacks are empty.

We will usually write a production rule p_B as $b_1 \wedge b_2 \ldots \wedge b_n \Rightarrow c$ where $b_i \in B$, $B \subseteq A$ and c is the symbol produced by p_B. We will denote step three of the ELPP as the sweep step. The EL-system can be used to represent the decontaminated vertices in a monk graph on each day.

Definition 15 (EL-system monk configuration)**.** *Let G be a monk graph where $n = |V(G)|$ and the vertices are labeled $0, 1, \ldots, n-1$. An EL-system $\phi = (A, \omega, P, \sigma)$ is monk configured of G if, and only if:*

- $A = V(G)$
- $\omega = \varnothing$
- $P = \{p_{x_0}, p_{x_1}, \ldots, p_{x_{n-1}}\}$ *where for* $0 \leq i \leq n-1$*:* $x_i = \{j \in A | j \to i \in E(G)\}$ *and* $p_{x_i}(A') = i$ *if, and only if,* $x_i \subseteq A'$*.*
- $\sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_p\}$ *contains the p pursuer strategies where the stack is in order (the first vertex to decontaminate is on top of the stack and the last at the bottom) and for each* $\sigma_i \in \sigma : |\sigma_i| = k$*. An entry E in the stack is empty, such that* $\{E\} = \varnothing$ *if the pursuer is idle.*

Lemma 1. *Let ϕ be a monk configured EL-system of a monk graph G. Let $S = \{S_0, S_1, \ldots, S_k\}$ be the set of states produced by ϕ. Then each S_i contains the decontaminated vertices on day i.*

Proof. We will do a proof by induction over $i, 0 \leq i \leq k$. First, we prove that the sweep step in the ELPP decontaminates the vertices swept by pursuers at any given day. Donote $S_i := S_i \cup \{E\}$ with the sweep operation. Since we do the sweep operation for all pursuers it is sufficient to prove its correctness for any of them. There are three cases:

1. The vertex E is contaminated. It should be decontaminated after the sweep operation. Proof: The union will include it in S_i.

2. The vertex E is decontaminated. It should remain decontaminated after the sweep operation. Proof: The union will not change S_i.

3. The pursuer is idle. S_i should remain unchanged. Proof: $\{E\} = \varnothing$ and thus S_i is set to $S_i \cup \varnothing$ which does not change S_i.

Note that the value of S_i does not change after the sweep step in the ELPP.

- **Let $i = 0$.** From step one in the ELPP: $S_0 = \omega = \varnothing$. All vertices are contaminated before the first sweep step, that is, there are no decontaminated vertices. Then the sweep step places the pursuers for day 0. All the vertices on top of the stack should be decontaminated according to definition 6. They will be decontaminated, which means S_0 will contain the vertices from on top of the stack since the sweep step is correct. The vertices from on top of the stack correctly corresponds to the first actions of the pursuers according to definition 15.

- **Let $i = x + 1$.** Assume S_x contains the decontaminated vertices on day x (induction hypothesis). All production rules are applied and a vertex v should only be in S_i if all vertices which have edges into v are decontaminated according to definition 6. From definition 15 we know that all produced values will be distinct since they correspond to $\{0, 1, \ldots, n-1\}$. Let $v \in S_i$. Let p_B be the production rule which produces v. Definition 13 in combination with that the direct decontamination of definition 6 has not happened yet implicitly states (indirect decontamination) that $v \in S_i$ if, and only if, the $B \subseteq A$ in p_B is a subset of S_x. From definition 15 B consists of the vertices which have edges into v. The insight is that $v \in S_i$ if, and only if, all vertices which have edges into v were decontaminated in the previous state, which is what we wanted to prove. Only the correct indirect decontaminations will occur since the induction hypothesis states that the previous state contains the correctly decontaminated vertices. Then, since the sweep step is correct and takes care of the direct decontamination of definition 6, S_i will contain only the decontaminated vertices.

□

If the EL-system production process terminates with $S_k = A$, the strategy is a winning strategy. However, we do not know if the strategy is optimal, there might still be a shorter strategy which also yields A.

Lemma 2. *Let ϕ be a monk configured EL-system of a monk graph G. Let $S = \{S_0, S_1, \ldots, S_k\}$ be the set of states produced by ϕ. Then if, and only if, the EL-system production process stops with $S_k = A$ the strategy is winning.*

Proof. From lemma 1 we have that S_k contains the vertices which are decontaminated. From definition 15 we have that $A = V(G)$. From the definition of a winning strategy all vertices must be decontaminated after all pursuer strategies have been applied in parallel, which is exactly what $S_k = A = V(G)$ corresponds to. □

Lemma 3. *If the input is finite for the EL-system production process and all operations are done in finite time, the EL-system production process terminates in finite time.*

Proof. The stacks have a finite length $k \geq 0$ and for each iteration of step two to five each stack has its size decreased by one. The stacks must then be empty after

k iterations which causes the EL-system production process to terminate. Since k is finite, the EL-system production process will terminate in finite time. □

Theorem 4 (EL-systems verify the monk problem)**.** *A monk configured EL-system can verify any finite strategy in finite time.*

Proof. We have that the EL-system production process will terminate from lemma 3. We have that it can determine whether the strategy was winning or not from lemma 2. □

3.1.1 A graphical representation

The EL-system allows for a compact and visually appealing representation of a strategy to the monk problem. In this section we will study the original formulation of the monk problem stated by Dilian Gurov. There exists exactly four optimal strategies, all guarantee that the monk will be found in six days or less regardless of how he moves. We will present one of the strategies below.

The five caves can be described with the directed graph in figure 3.1. A vertex corresponds to a cave and an edge in the graph indicates that a movement is possible between two caves. Each cave is labeled with a number one to five. Given this, a pursuer strategy can be described as [2, 3, 4, 4, 3, 2]. This means that if we look in the cave labeled with two the first day, then the cave labeled with three the second day and so on, we will eventually find the monk in at most six days. A way of visualising the strategy is to draw a timeline as in figure 3.2.

The timeline can be interpreted like this. The graph on row i represent the state on day $t = i$. A vertex is coloured with red if we choose to directly decontaminate this cave on day i. The vertices coloured with either red or yellow are decontaminated vertices and the goal is to colour all vertices in as few steps as possible. If a vertex is decontaminated, all outgoing edges from this vertex are *blocked*, such edges are coloured in red. If all incoming edges to a vertex are blocked, the vertex is considered indirectly decontaminated on the next day and will be coloured in yellow.

Suppose we know that vertices two and four are decontaminated on day i. We can then conclude that vertex three is decontaminated on day $i+1$ because the only way to get to vertex three is by going from vertex two or four, and we knew that the monk was not there, hence he cannot be in vertex three during the next day.

In the sample strategy we choose to decontaminate vertex two at day one, which means that the edges $2 \to 1$ and $2 \to 3$ will be blocked. At day two, the only incoming edge to vertex one is blocked, so this vertex is decontaminated. However, none of

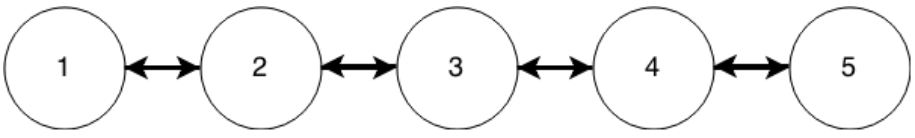

Figure 3.1. The graph representation of Dilian Gurov's monk problem.

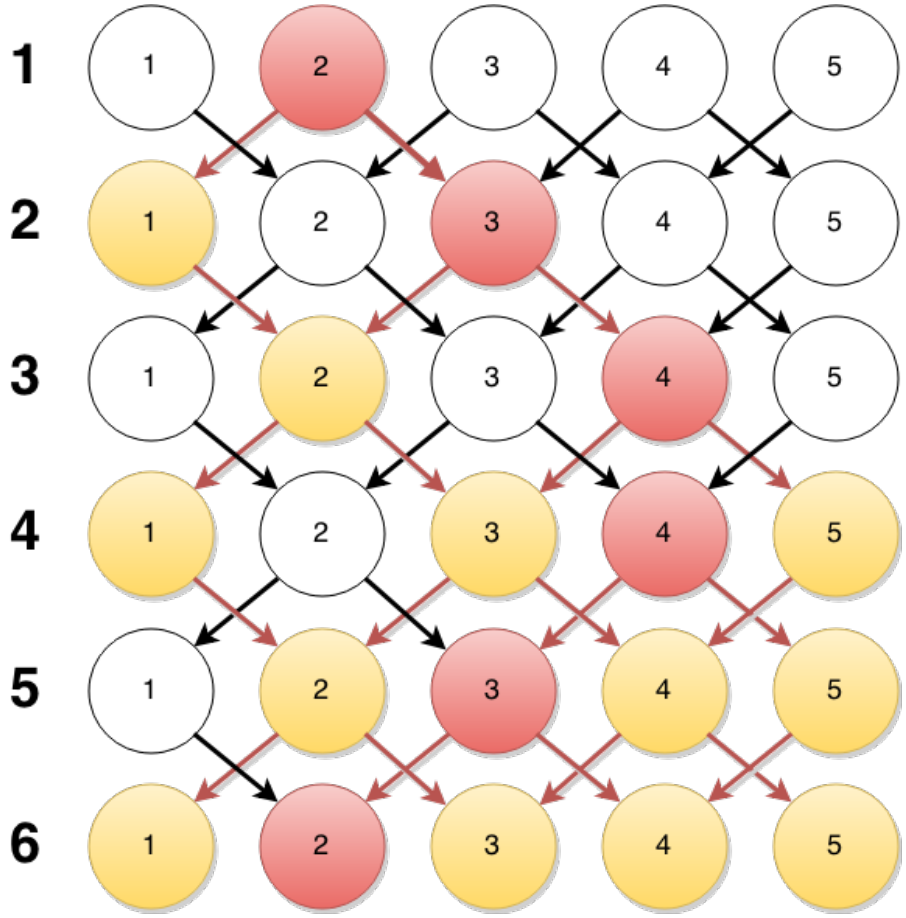

Figure 3.2. One of the four optimal strategiess for the monk problem visualised with a timeline. The set of decontaminated vertices are marked with yellow or red. On day six, the set of decontaminated vertices comprises the whole set of vertices, and if the monk exists, he must have been found.

the other vertices are blocked, for example it is still possible to come to vertex three during day two if the monk is in vertex four on day one. The second day we choose to decontaminate vertex three which will block all incoming edges to vertex two on day three. Note that on day three, vertex one is contaminated because the monk could have followed the path $3 \to 2 \to 1$. However, if continuing to decontaminate the caves [4, 4, 3, 2] all possible paths starting in any of the five vertices at day one will be cut off, and we can conclude that the monk will be found in at most six days.

In the EL-system for the monk problem we have three production rules:

- $2 \Rightarrow 1$ *(1)*
- $4 \Rightarrow 5$ *(2)*
- $n - 1 \wedge n + 1 \Rightarrow n$ where $1 \leq n \leq 5$

The last rule can be explicitly written as

- $1 \wedge 3 \Rightarrow 2$ *(3)*
- $2 \wedge 4 \Rightarrow 3$ *(4)*
- $3 \wedge 5 \Rightarrow 4$ *(5)*

If we let the alphabet for the monk problem consist of the symbols 1 to 5 and we input the pursuer strategy [2, 3, 4, 4, 3, 2], iterating the EL-system production process gives the following:

- $S_0 = \varnothing \cup \{2\}$ (pop 2)
- $S_1 = \{1\} \cup \{3\}$ (use rule *1* and pop 3)
- $S_2 = \{2\} \cup \{4\}$ (use rule *3* and pop 4)
- $S_3 = \{1, 3, 5\} \cup \{4\}$ (use rule *1*, *4*, *2* and pop 4)
- $S_4 = \{2, 4, 5\} \cup \{3\}$ (use rule *3*, *5*, *2* and pop 3)
- $S_5 = \{1, 3, 4, 5\} \cup \{2\}$ (use rule *1*, *2*, *4*, *5* and pop 2)

3.1.2 Matrix representation of the EL-system

For implementation purposes, we need an efficient representation of the EL-system given a graph G with m vertices. This is possible by using an $m \times m$ matrix P which represents the production rules. A production rule $a_1 \wedge a_2 \ldots \wedge a_n \Rightarrow B$ corresponds to putting a single one in columns $a_1, a_2, \ldots, a_n$ on row B. If each vertex in the graph is labeled with a number $1 \leq i \leq m$; then the set of decontaminated vertices is represented by a column vector s with m elements where the i:th element is a single one if vertex number i is decontaminated, or zero if vertex i is contaminated.

Observation 1. *P is the adjacency matrix of G.*

The vertices being directly decontaminated by pursuers during day t are represented by a column vector r with m elements. The i:th element in r is a single one if it is being directly decontaminated by a pursuer or a single zero otherwise.

The set of decontaminated vertices on day $t+1$ is given by the formula

$$s_{t+1} = P \odot s_t \uplus r$$

Two mathematical operations called reduced multiplication (*rmul*), denoted with $\odot$ and reduced addition (*radd*), denoted with $\uplus$ are used. These two operations are described in psuedo C-code below:

```
// P is an m*m matrix and s is an m*1 column matrix
matrix rmul(matrix P, matrix s) {
    matrix c, w; // c and w are both m*1 column matrices
    for (int row=0; row<m; row++) {
        for (int column=0; column<m; column++) {
            if (P[row][column]==1) {
                // w[i] should contain the Hamming
                // weight for row i in the matrix P
                w[row]++;
            }
        }
    }
    c=P*s; // perform matrix multiplication
    for (int i=0; i<m; i++) {
        if (c[i]>0) {
            if (w[i]==c[i]) {
                c[i]=1;
            } else {
                c[i]=0;
            }
        }
    }
    return c;
}
// both s and r are m*1 column matrices
matrix radd(matrix s, matrix r) {
    for (int i=0; i<m; i++) {
        s[i]+=r[i];
        if (s[i]>1) {
            s[i]=1;
        }
    }
    return s;
}
```

The matrix representation for each of the steps in fig 3.1 looks like this:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \odot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \uplus \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \odot \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \uplus \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \odot \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \uplus \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \odot \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \uplus \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \odot \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} \uplus \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \odot \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \uplus \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

The correctness of the verifier can be motivated as follows: The state of each vertex v (contaminated or decontaminated) on day $t+1$ is determined by matrix multiplication of row v in P with the column vector s. The requirement for v to be decontaminated is that all vertices with an endpoint in v are decontaminated on day t. The intermediate value in $c[v]$ corresponds to the number of vertices with an endpoint in v which are decontaminated on day t. Since the sum $c[v]$ only gets incremented when there exists an edge $i \to v$ ($P[v][i] = 1$) and vertex i is decontaminated ($s[i] = 1$), it follows that if the sum is equal to $w[v]$ (the Hamming weight of row v), then all vertices which has an edge to v are decontaminated, and thus v will be decontaminated on day $t+1$, which corresponds to $c[v]$ being equal to one when *rmul* returns.

If the number of vertices in the graph does not exceed the number of bits which can fit in a register, it is possible to calculate the next state in linear time. A row in the matrix P and the state s can be represented as integers. The *rmul* operation can be translated to *bitwise and*, and *radd* can be translated into *bitwise or*. A bitcounter is used to check if the strategy decontaminates all vertices in graph. More precisely, let the matrix P and the strategy A be represented as an array of integers. Each integer can be seen as a bitvector B with REG_SIZE bits. The state is also represented as a bitvector with REG_SIZE bits, and bit b in the next state on round i is calculated by performing bitwise and with $P[b]$ followed by bitwise or with $A[i]$. The criterion for decontamination is checked each time a vertex is decontaminated by comparing a bitcounter bc with the number of vertices in the graph (NUM_NODES).

```
REG_SIZE = sizeof(int)
#define BITMASK(i) 1 << (REG_SIZE - (i))

assert(NUM_NODES <= REG_SIZE)
s, sn = 0; // current and next state
bc = 0;
for (i = 0; i <= A.length; i++) {
    /* rmul */
    for (int j = 0; j < NUM_NODES; j++) {
        if (P[j]&sn == P[j]) {
            s |= BITMASK(j) // set bit j
            if (++bc == NUM_NODES) return "OK"
        }
    }
    /* radd */
    s |= A[i]

    sn = s;
    s = 0;
    bc = 0;
}

return "NOT OK"
```

3.2 Problem complexity

To be able to use an EL-system as a polynomial time verifier, we need to make the assumption that any optimal strategy is polynomial in size.

Conjecture 5. *The length of any optimal strategy for a graph G is bounded by $P(|V(G)|)$ where P is a polynomial.*

Corollary 5.1. *MSNDP and MSLDP are in NP.*

Proof. Given an $m \times m$ adjacency matrix representing a matrix G and a strategy S, we can determine if the strategy is winning using an EL-system.

Calculating the Hamming weight for each row in the adjacency matrix can be done in $O(m^2)$ time, note that this only needs to be done once. Matrix multiplication in *rmul* is also done in $O(m^2)$ and *radd* goes in linear time, which means that calculating s_{i+1} can be done in $O(m^2)$. If we assume that all vertices are decontaminated on day $t = n$, the time complexity of verifying a strategy is $O(nm^2)$, which is polynomial with respect to input size.

The strategy S given as a witness for MSNDP(G, k) must be checked so that it uses at most k pursuers. The k-value is the maximum number of pursuers used in any of the i steps, and can be calculated in linear time (by counting the number of ones in the column matrix r). The answer to *MSLDP(G, k, l)* is *YES* if and only if the length of S is equal to l and the number of pursuers in the strategy is equal to k. The length of the strategy is calculated by counting the number of iterations in the EL-system production process which takes no extra time and the number of pursuers can be determined in the same way as for MSNDP. Since we assume (see conjecture 5) that the length of the strategy is polynomial of the size of the graph, this process runs in polynomial time. □

Conjecture 6. *MSNDP is NP-hard.*

Corollary 6.1. *MSLDP is NP-hard.*

Proof. We can reduce MSNDP to MSLDP by letting MSLDP check for a strategy of any length. If a strategy of length l is winning, then a strategy with length $l + i$ $i > 0$ must be winning too, since pursuers can stay idle for i days. The length of a strategy for a graph with n vertices is bounded by the number of possible states for the graph. Each vertex can be either contaminated or decontaminated, thus the number of states becomes 2^n. Hence, the reduction is a simple call to MSLDP with $l = 2^n$.

```
1 MSNDP(G, k)
2     return MSLDP(G, k, 2^|G|)
```

□

Corollary 6.2. *MSNDP and MSLDP are NP-complete.*

Proof. If MSNDP and MSLDP are NP-hard, it remains to be shown that a solution to MSNDP and MSLDP can be verified in polynomial time, which is shown in corollary 5.1. □

3.3 Stable component decomposition

In this section we introduce stable components and their relevance to the monk problem. The decomposition simplifies the problem and the different parts can be solved separately which provides the possibility to make parallel computations.

There might exist some parts of the graph, which once decontaminated will remain decontaminated forever, regardless of how the evader moves (see fig 3.3). We call such a part of the graph a stable component.

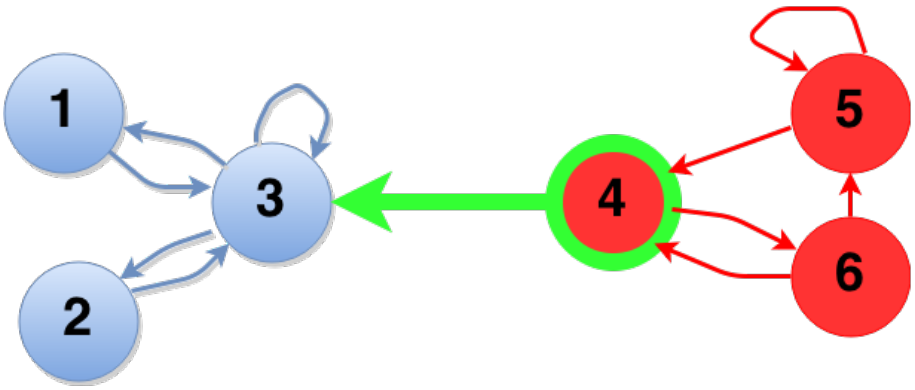

Figure 3.3. The vertices of this graph can be partitioned into one red set and one blue set using the stable decomposition algorithm. This works because once an evader has traveled along the edge $4 \to 3$ she can no longer come back to the red set. Thus, we can focus on finding a strategy for decontaminating the red set first, and then move on to decontaminate the blue set. The vertex with a green circle is called a transit vertex and the green edge is called a transit edge.

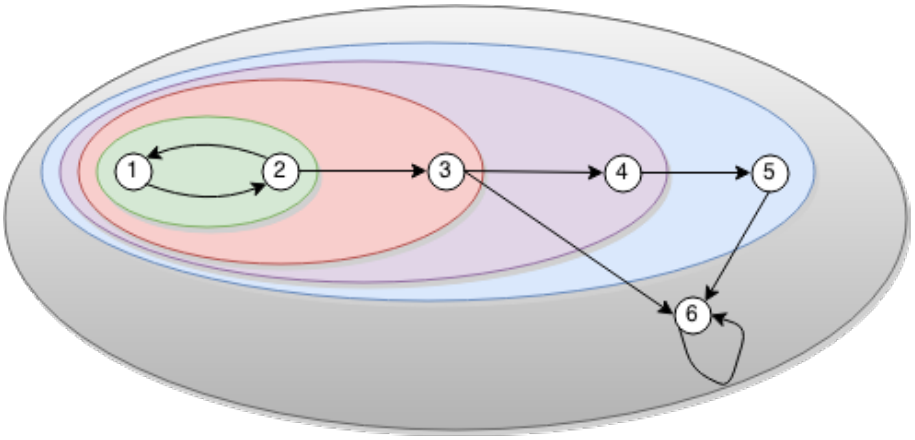

Figure 3.4. An example of all the stable components of a graph. There is a stable component inside each ring. Note that there are no incoming edges into any ring from outside of it.

Definition 16 (Stable component)**.** *Given a monk graph M, a stable component S is a non-empty subgraph of M where:*

- *there are no edges in M which start from a vertex outside of S, and end in a vertex in S. More formally:* $\neg\exists(u \to v) \in E(M) : v \in V(S) \wedge u \in V(M) \backslash V(S)$
- *S consists of exactly one component*

An example showing all the stable components of a graph can be seen in fig 3.4.

Theorem 7. *If S is a stable component of the monk graph G and all vertices in S are decontaminated on day t_i then they will remain decontaminated on day t_{i+1}.*

Proof. Assume that $v \in S$ is recontaminated on day t_{i+1}, then the recontamination source must come from a vertex $u \notin S$ since all vertices in S are decontaminated on day t_i and decontaminated vertices can not recontaminate any vertices. But if $u \notin S$ recontaminate v there must exist an edge $u \to v$, which contradicts the claim that S is a stable component of G. Thus all $v \in S$ remain decontaminated on day t_{i+1}. □

Theorem 7 is the basis of stable component decomposition, if a set of vertices can not be recontaminated we can disregard the subgraph consisting of those vertices and focus on finding a search strategy for the remainder of the graph.

Proposition 1. *A monk graph is a stable component of itself.*

Proof. Since for the whole *monk graph* $V(M) = V(S)$ we will have $V(M) \backslash V(S) = \varnothing$ and thus there can be no incoming edges to M. M consists of one component per definition, and thus, M is also a stable component. □

Lemma 8 (No return property)**.** *Let S be a stable component of the monk graph M and let v be a vertex outside of S reached by following an edge from a vertex in S. There does not exist a walk from v into a vertex in S. More formally: $\forall u \to v \in E(M) \neg\exists w \in V(S) : Walk(v, w)$ where $u \in V(M), v \in V(M) \setminus V(S)$ and $Walk(x, y)$ is the predicate which is true if, and only if, there exists a walk in M starting in x and ending in y.*

Proof. Assume there exists such a walk. The walk starts outside of S and then eventually enters S. To do this there must exist an edge $v \to u \in E(M)$ where $u \in S$ and $v \in V(M) \setminus V(S)$. Such an edge implies that S is not a stable component, causing a contradiction. Thus, no such walk can exist. □

Definition 17 (Stable reduction)**.** *Let M be a monk graph. $R = \{(R_0, S_0), (R_1, S_1), \ldots, (R_k, S_k)\}$ is a stable reduction of length $k+1$ if, and only if, $R_0 = M$ and S_j is a stable component of R_j and R_{i+1} is the graph obtained by removing S_i from R_i, $0 \leq i < k, 0 \leq j \leq k$. No S_j is the empty graph. The result of removing S_k is the empty graph.*

Stable components become more useful when defining a minimal stable component. A minimal stable component can not be decomposed into smaller stable components.

Definition 18 (Minimal stable component)**.** *A minimal stable component S_{min} of a monk graph M is a stable component of M which can not be decomposed into smaller stable components. More formally: there does not exist a stable component S' of S_{min} where $V(S') \subset V(S_{min})$.*

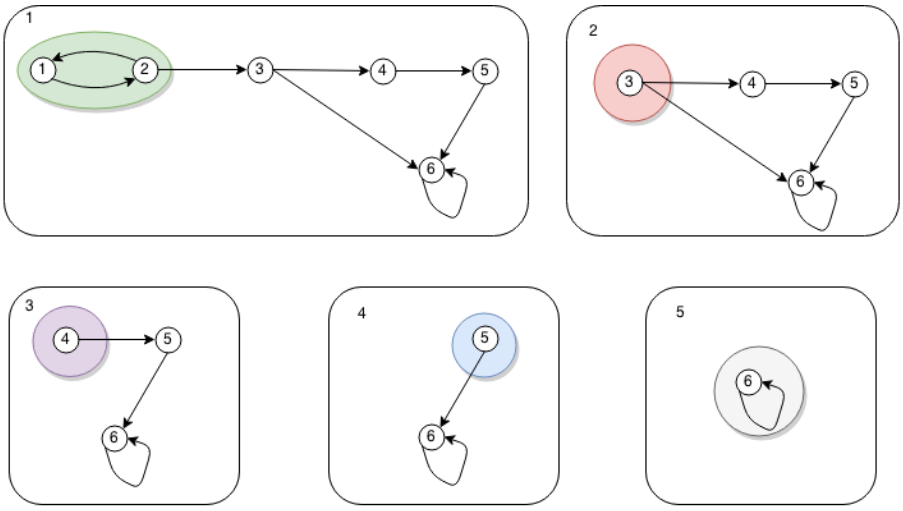

Figure 3.5. An example of a minimal stable reduction $R = \{(R_0, S_0), (R_1, S_1), \dots (R_4, S_4)\}$. R_{i-1} is the graph in box i and S_{i-1} is the subgraph in the ball of box i, $1 \leq i \leq 5$. The subgraph S_{i-1} is removed at each stage and the remaining graph is R_i.

Definition 19 (Minimal stable reduction)**.** *A minimal stable reduction is a stable reduction* $R = \{(R_0, S_0), R_1, S_1), \dots (R_k, S_k)\}$ *where for* $0 \leq j \leq j$*:* S_j *is a minimal stable component of* R_j*.*

An example of a minimal stable reduction can be seen in figure 3.5.

Lemma 9 (Stable decomposition)**.** *If* $R = \{(R_0, S_0), R_1, S_1), ..., (R_k, S_k)\}$ *is a minimal stable reduction of* M *then* $\{V(S_0), V(S_1), ..., V(S_k)\}$ *forms a partition of* $V(M)$*.*

Proof. We will prove that the parts are disjoint and that all vertices can be found in the partition. Assume there exists two i, j such that $V(S_i) \cap V(S_j) = C \neq \varnothing$, then $v \in C$ must exist, but one of S_i and S_j must have come first in the stable reduction and thus removing v from the remaining monk graphs in the reduction, which means that whichever came second could not contain v. Because of this contradiction, the negative must hold; $\forall i, j : V(S_i) \cap V(S_j) = \varnothing$, that is, all parts are disjoint.
Assume that $\exists v \in M \forall i : v \notin V(S_i)$. This implies that v is still left after the reduction, which contradicts that the result is an empty graph. Therefore, the negative must hold: $\forall v \in M \exists i : v \in V(S_i)$. □

Lemma 10 (Stable reduction strategy)**.** *A stable reduction* $R = \{(R_0, S_0), R_1, S_1), \dots, (R_k, S_k)\}$ *of a monk graph* M *forms a decontamination sequence* $Q = \{S_0, S_1, ..., S_k\}$ *which guarantees that* M *can be cleaned by decontaminating* Q *in order.*

Proof. We can assume we have a winning strategy W_j for each S_j. The total strategy is $[W_0, W_1, ..., W_k]$, $0 \leq j \leq k$. We will now show that the total strategy is winning. For each i: S_i have no incoming edges from contaminated vertices outside of S_i,

if it did, it would not be a stable component of R_i. This implies the only vertices which can recontaminate S_i is in S_i itself, thus S_i can be decontaminated by W_i without risk of recontamination. Note that after W_i has decontaminated S_i, S_i will be decontaminated during the remaining days. Since the vertices of the subgraphs form a partition of M, all vertices in M will be decontaminated by the total strategy. Therefore, the total strategy is winning. □

Theorem 11 (Stable reduction search number)**.** *Let* $R = \{(R_0, S_0), R_1, S_1), ..., (R_k, S_k)\}$ *be a minimal stable reduction of a monk graph* M*. The search number of* M *is equal to the maximum search number of* $\{S_0, S_1, \ldots, S_k\}$*.*

Proof. Let Z_i be the search number of S_i. Observe $(R_i, S_i) \in R$, there is no way to decontaminate S_i from outside of S_i since there are no edges from contaminated vertices outside of S_i which end in S_i. Therefore, the only way to decontaminate S_i is to use a winning strategy for S_i, and the optimal one uses Z_i pursuers. Let W be a total winning strategy, Z be the search number of M, and Z_{max} be $max(\{Z_i | 0 \leq i \leq k\})$. Assume that $Z < Z_{max}$ and let $S_{max} \in \{S_i | 0 \leq i \leq k\}$ be a subgraph which has search number Z_{max}, then S_{max} can not be cleaned with only Z pursuers and thus W can not be a total winning strategy. Therefore, $Z \geq Z_{max}$. Since a monk graph can be decontaminated by any amount of pursuers larger than or equal to its search number, Z_{max} is also a sufficient limit for each S_i. Thus, $Z = Z_{max}$. □

The properties of a minimal stable component can be found in a strongly connected component.

Theorem 12. *Let* $R = \{(R_0, S_0), R_1, S_1), ..., (R_k, S_k)\}$ *be a minimal stable reduction of a monk graph* M*. The set* $Q = \{S_i | 0 \leq i \leq k\}$ *is equal to the strongly connected components of* M*.*

Proof. We will first prove that the components in Q are strongly connected. Assume S_i is not strongly connected. Then there must exist two distinct vertices $u, v \in S_i$ such that there exists no walk in S_i from u to v or there exists no walk in S_i from v to u. Assume neither walk exists, then u and v belong to different components, which contradicts that S_i is a stable component, so at least one of the walks must exist.

Assume it is a walk from u to v that exists. We will show that a walk v to u must also exist. We know there exists an edge into v since there is a walk from u to v. Assume only the incoming edge/edges exists and no outgoing walks, then the subgraph S_i' of S_i with v removed is a stable component of S_i and $V(S_i') \subset V(S_i)$, which contradicts that S_i is a minimal stable component. Therefore, there must also exist a walk v to w where $w \in V(S_i) \setminus v$.

Let $T_x = \{w \in S_i | Walk(x, w)\}$, we have showed that T_v is not empty. Assume $u \notin T_v$. Then the subgraph consisting of the vertices in $T_u \setminus T_u \cap T_v$ form a smaller stable component of S_i which contradicts that S_i is a minimal stable component. Thus, $u \in T_v$, that is, there exists a walk from v to u. S_i is strongly connected.

We will now show that each strongly connected component in Q can not be expanded. Assume S_i can be expanded by adding a vertex $x \in V(M) \setminus V(S_i)$. According to lemma 9, x must either belong to S_h or S_j where $h < i < j$. We also know implicitly from definition 17 that $V(S_i) \subset V(R_h)$ and $V(S_j) \subset V(R_i)$. Assume $x \in S_h$, then S_h is not a stable component since there exists an edge in R_h in the walk from S_i into x. Therefore, $x \notin S_h$. Assume $x \in S_j$, then S_i is not a stable component since there exists an edge in R_i in the walk from x to S_i. Therefore, $x \notin S_j$. The contradiction implies that S_i can not be expanded. Thus, Q consists of the strongly connected components of M. □

3.4 Cycle decomposition

In this section we will propose a decomposition for G based on cycles. The decomposition provides an algorithm for finding a strategy in linear time for certain subgraph structures, and for splitting strongly connected components by guarding. The cycle decomposition works as follows:

1. Start by identifying all elementary circuits (cycles) C in the graph
2. Build a new undirected graph G' as follows: Let each cycle in C be a vertex in G'. Create an edge $c_1 \longleftrightarrow c_2$ if $c_1 \cap c_2 \neq \varnothing$.
3. Denote a component in G' with $D_{G'}$. If $D_{G'}$ consists of the cycles $d_1, d_2 \ldots d_n$ then we form a *part* of the partition of G as $V(d_1) \cup V(d_2) \cup \ldots V(d_n)$.
4. Each vertex in G which does not belong to a component in G' is itself a part of the partition of G. Such a part is called a simple part.

There exists a trivial strategy if the cycle decomposition of a stable component produces a clique, and all cycles share a common element.

Theorem 13 (Clique theorem)**.** *If G' is the graph obtained by running the cycle decomposition algorithm on a a monk graph G, $P = V(d_1) \cup V(d_2) \cup \ldots V(d_n)$ is the part formed from the component $D_{G'} \subseteq G'$, and we define C_{max} as the number of vertices in the largest cycle in $D_{G'}$; then if $D_{G'}$ is a clique, and $P = V(d_1) \cap V(d_2) \cap \ldots V(d_n) \neq \varnothing$, we can decontaminate P by staying at one of the nodes in $V(d_1) \cap V(d_2) \cap \ldots V(d_n)$ for exactly C_{max} days.*

Proof. The cut $R = V(d_1) \cap V(d_2) \cap \ldots V(d_n)$ contains the elements which all cycles in the graph has in common. Let $R_g \in R$ be the element being guarded. Since R is not empty per definition, R_g must exist. Given that an evader is forced to move on each day and that the graph is finite, the evader must eventually complete a cycle C in the graph. R_g is a vertex which belongs to all cycles in the graph, so $R_g \in C$. Since the evader has visited each vertex in C, she must also have visited R_g, where the evader was captured.

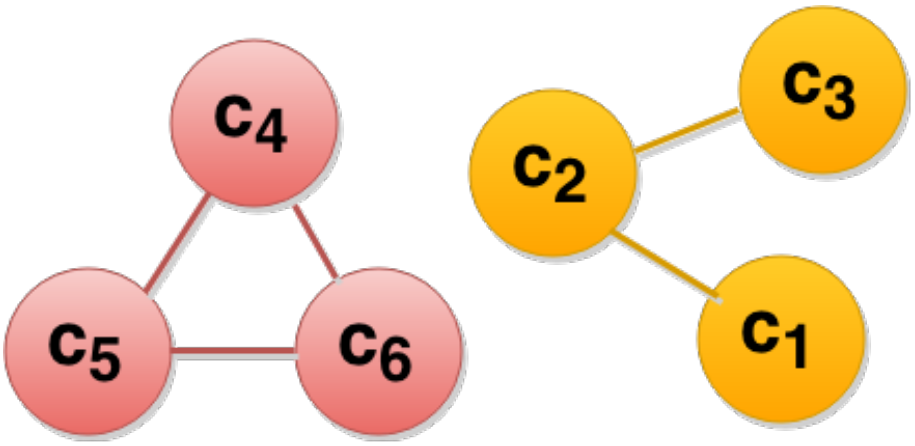

Figure 3.6. An example of a graph G' obtained by running the cycle decomposition algorithm on a graph G. Each vertex represents a cycle in G and an edge between two vertices means that the cycles have a common element.

From here is follows that an evader must have been captured after a cycle has been traversed. The largest cycle has length C_{max}, and one vertex is visited on each day; thus the longest path which an evader can follow without visiting R_g must have length $C_{max} - 1$. As such it suffices to stay at R_g for exactly C_{max} days. □

For each component in G' which is not a clique, the strategy is non-trivial.

Here follows an example of how the cycle decomposition algorithm works. The graph which is going to be partitioned is shown in fig 3.3.

The cycles of this graph are $c_1 = [4, 6]$, $c_2 = [4, 5, 6]$, $c_3 = [5]$, $c_4 = [1, 3]$, $c_5 = [2, 3]$ and $c_6 = [3]$. The graph G' will consist of two components $D_1 = [c_1, c_2, c_3]$ and $D_2 = [c_4, c_5, c_6]$ as shown in fig 3.6.

We can now partition the vertices of G in the parts $P_1 = [4, 5, 6]$ and $P_2 = [1, 2, 3]$. Since the component D_2 is a clique we can decontaminate P_2 by looking twice at vertex 3 ($c_4 \cap c_5 \cap c_6$).

3.5 Lower and upper bounds

To avoid looking for a strategy with k pursuers when none exist, it is of great interest to narrow the search space by establishing lower and upper bounds for the search number. In this section we give examples of a couple of different techniques on how to do this for a stable component G.

Definition 20 (Lower bound)**.** *A lower bound l for a monk graph G is an integer, such that there exists no optimal strategy for G using less than l pursuers.*

Definition 21 (Upper bound)**.** *An upper bound u for a monk graph G is an integer, such that any optimal strategy utilises at most u pursuers.*

Proposition 2 (Lower bound using minimum indegree)**.** *Let G be a monk graph. Then minimum indegree x of the vertices in G is a lower bound for G.*

Proof. We observe the two parts of definition 6. Let s be the search number of G. Note that no indegree can be greater than $|V(G)|$, since that implies that there are more vertices in the graph than $|V(G)|$. Assume $s < x$. If $s = V(G)$, then $x > |V(G)|$ which leads to a contradiction. We know that $s \leq |V(G)|$ from proposition 4, thus $s < |V(G)|$. This implies that a strategy must be longer than one day, since all vertices can not be directly decontaminated during the first day. Therefore, atleast one vertex must be indirectly decontaminated for the number of decontaminated vertices to increase between two days. Indirect decontamination of a vertex v requires that all vertices which have incoming edges into v are decontaminated in the previous day. Since x is the minimum indegree, we need atleast x pursuers to directly decontaminate the vertices which have incoming edges into v. Since $s < x$, s pursuers can not decontaminate the graph, which contradicts that s is the search number. Therefore, the assumption must be wrong and $x \leq s$, that is, the minimum indegree is a lower bound for G. □

Proposition 3 (Lower bound using cliques)**.** *If G contains a clique with n vertices, then $n - 1$ is a lower bound for G.*

Proof. An evader residing in one of the vertices in the clique can go to one of $n - 1$ vertices during his next turn and still remain in the clique. Thus, to force the evader out of the clique, we must use exactly $n - 1$ pursuers, unless each of the vertices in the clique has a self-loop, in which case n pursuers are required. □

Proposition 4 (Trivial upper bound)**.** *Let G be a monk graph. Then $|V(G)|$ is a upper bound of G.*

Proof. All vertices in G can be decontaminated in one day by direct decontamination, that is, placing a pursuers on each vertex. □

We can also use the cycle decomposition G' to find an upper limit for G.

Observation 2 (Reduction of G')**.** *We can remove an edge $u \leftrightarrow v$ in the graph G' by guarding the vertices $V(u) \cap V(v)$.*

If we remove edges from G' by guarding vertices in G as described above, we obtain a reduction of G'. For the ease of notation, such a reduction is denoted by G^- and is defined by a sequence of reductions $s_1, s_2 \dots s_n$ where each s_i is a removal of an edge in G'. The number of vertices that must be guarded in G at step i is denoted with $|s_i|$. See fig 3.8 for an example. From here follows the proposition we are looking for.

Proposition 5 (Upper limit using cycle decomposition)**.** *If $s_1, s_2 \dots s_n$ is the sequence used to receive the reduction G^- with each component in G^- being a clique with each cycle in the clique having a common element, an upper bound for G can be expressed as $1 + \sum_{i=1}^{n} |s_i|$.*

Proof. Since the cycle decomposition partition all vertices in G, it follows that any vertex in G must either belong to a clique in G' (equivalent to a component and clique in G^-), or being guarded due to the reduction of G'. If an evader resides in any vertex being guarded, they are captured immediately. It is not possible to move between two cliques in G' since any such connection must be guarded. Thus any evader must reside in the vertices of G corresponding to a clique in G^-. These cliques can be decontaminated in sequence according to the strategy which follows from theorem 13. Since it is impossible for an evader to move between the cliques in G', considering the positioning of the $\sum_{i=1}^{n} |s_i|$ guards, we can inspect each clique separately, using exactly one pursuer. From here the result follows. □

We end this section by giving an estimate of the search number.

Definition 22 (Edvin's estimate)**.** *Let $I(v)$ be the indegree of v, that is $I(v) = |\{u \in V(G) | u \to v \in E(G)\}|$. Let I_{max} be the maximum indegree of all nodes $v \in V(G)$.*

Edvin's estimate is the smallest number x for which there exists a path $v_0, v_1 \dots v_k$ of distinct vertices; such that for all vertices v_i, $0 \leq i \leq k$ in the path, it holds that $I(v_i) \leq x + i$ and $I(v_k) = I_{max}$.

The value x in Edvin's estimate can be found efficiently using *depth-first traversal* of G. Here is an example of how such an algorithm can be implemented:

```
ESTIMATE(G)
    // Sort vertices in ascending order based on indegree
    sort_ascending(vertices)
    for v_0 in vertices (ascending order)
        x = indegree[v_0]
        if validPathDFS(v_0, x, 0, visited, indegree)
            estimate = x
    return max(1, x)

validPathDFS(v_i, x, i, visited, indegree)
    if visited[v_i] or indegree[v_i] > x + i
        return false
    if indegree[v_i] = max(indegree)
        return true

    visited[v_i] = true

    foreach neighbour to v_i
        if DFS(neighbour, x, i+1, visited, indegree)
            return true

    visited[v_i] = false
    return false
```

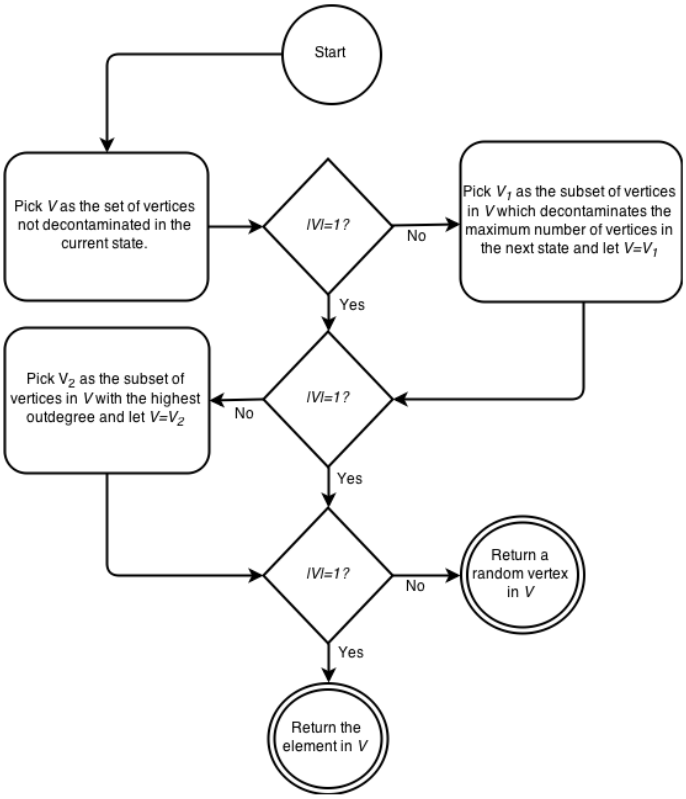

Figure 3.7. A greedy algorithm which selects a vertex to decontaminate.

3.6 A heuristic approach

It is still unknown if there exists a polynomial-time algorithm for MSLDP and MSNDP. In this section we will try to find a non-exact solution to MSLDP and MSNDP based on heuristics. The selection process is based on a greedy algorithm, which selects a vertex v to decontaminate based on the schema in 3.7.

The heuristics starts with the *the partition step*. The partition step is based on minimal stable components (definition 18.) The subgraph induced by the vertices of each partition is a strongly connected component (theorem 12).

The heuristics process the strongly connected components S, either one by one or in parallel. The components with the highest search number should be solved first to minimize the length of the solution due to theorem 11. We propose a calculation of the search number using Edvin's estimate to determine the order of the components in S.

The search step is executed for each component in S and consist of a search for a winning strategy by repeatedly execute a method *SOLVE*. It implements a recursive algorithm which selects a set of vertices to decontaminate at each step in the algorithm. The vertices to be decontaminated is given by a *selector*. The

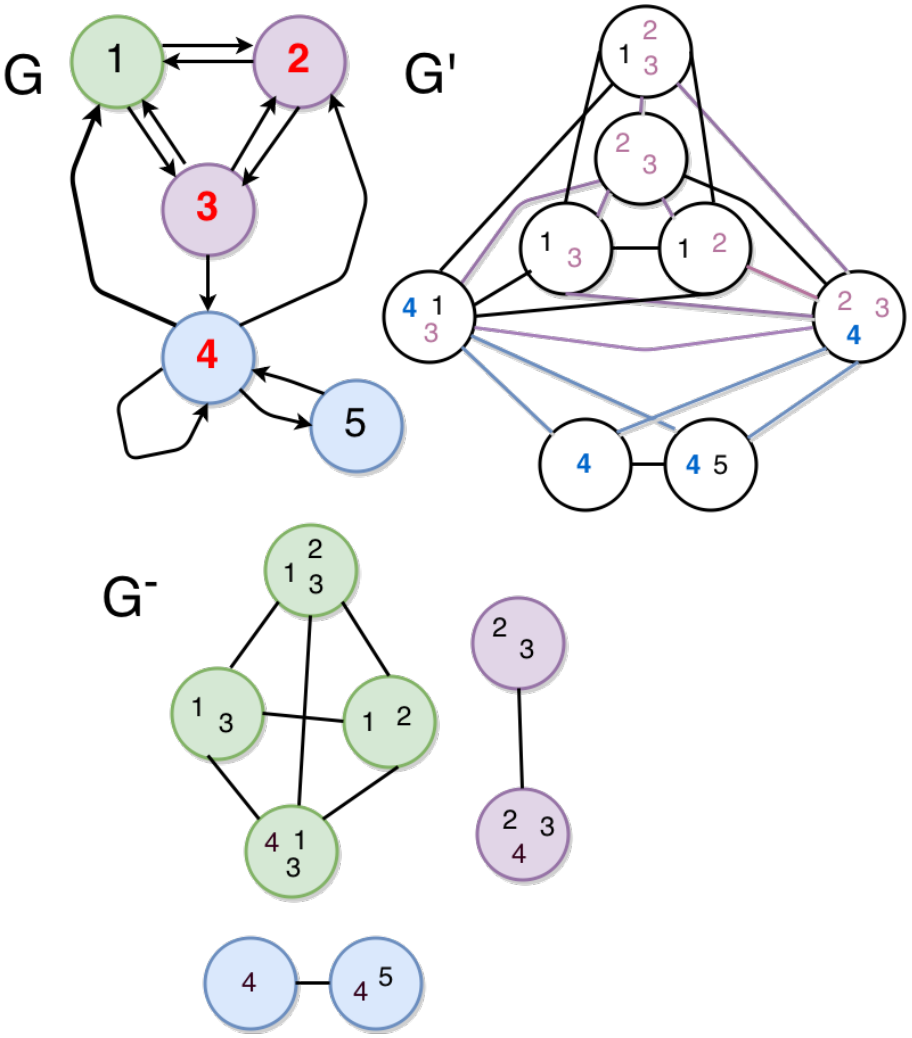

Figure 3.8. A monk graph G, the corresponding cycle decomposition G', and a reduction G^- yielding an upper limit of four. The graph G^- is given by removing the edges in G' marked with blue to isolate the cycle [4, 5] and put a guard on vertex four. The blue component is a clique with a common element (vertex four marked with blue) and can be solved easily by theorem 13. Since the remaining vertices form a clique in G' *without* a common element, it needs to broken down in smaller pieces. This is done by removing the edges marked in lilac, positioning guards at vertex two and three. A non-optimal strategy derived from this decomposition is to stay at vertices [1, 2, 3, 4] at day one, followed by vertices [2, 3, 4, 5] at day two. The optimal strategy uses two pursuers.

selector we propose is based on a greedy algorithm which maximises the number of decontaminated vertices on the next day. The search can be, for example, a linear search, or a binary search. Once a winning strategy has been found, we continue to process the next component in S. The bounds used in our binary search was the minimum indegree as lower bound (proposition 2) and the number of vertices as upper bound (proposition 4). If an ordinary binary search is used, the search step has to be repeated $O(\log(upper - lower))$ times, which is the best possible asymptotic behavior.

The heuristics ends with *the merge step* which begins when all components in S have been solved. In the merge step, the solutions for each of the strongly connected components are put together to form a solution for the whole graph (lemma 9). The search number for the whole graph is given by the maximum search number for any of its strongly connected components (theorem 11).

The order for how the strategies should be put together is defined by the *topological order* of a DAG D where each vertex $S_1, S_2 \dots S_n$ in D is a strongly connected component. An edge $S_m \to S_n$ in this DAG, means that there exists a vertex $u \in V(S_n)$ with an endpoint in $V(S_m)$. Such an edge is called a *transit edge* and the u is called a *transit vertex*. A topological order for D can be expressed as a sequence $S_1, S_2 \dots S_n$ of minimal stable components, as described in a stable reduction (see definition 17).

Proposition 6. *The heuristics using: a selector which picks only one vertex, and binary search can be implemented with time complexity $O(|V|^2 \log |V| * K * L * |E|)$, where L is the bounded length of a solution and K is the number of pursuers used in the strategy.*

Proof. The search step is repeated $O(\log |V|)$ times. In each search step we do the following work at most $K * L$ times:

1. **Check if current state is visited and mark a state as visited** This can be done in $O(|V|)$ time by finding all decontaminated vertices. The vertices are hashed and checked against a bloom filter which goes in $O(1)$ time.

2. **Check if current state is winning** This is done in $O(|V|)$ by counting the number of decontaminated vertices.

3. **Select the vertex to decontaminate** For each vertex: calculate the number of decontaminated vertices in the next state (select candidate vertices). If the graph representation contains indegree for each vertex and the vertices are stored in an adjacency list, the time complexity for finding candidate vertices is $O(|V| + |E|)$. The candidate vertices are added to a maxheap which takes $O(|V| \log |V|)$ whereby we pick a random element in $O(1)$ time. Thus, the time complexity for this step is $O(|V| \log |V| + |E|)$.

4. **Calculate current and next state** Copying the states goes in $O(|V|)$ time.

5. **Decontaminate the vertex selected by the selector** The indegree for each neighbour to the vertex selected must be reduced by one, which goes in $O(|E|)$ time.

6. **Calculate the next state (transition)** Restoring indegree for each vertex goes in $O(|V|)|$ time. Decontaminating the vertices from the current state goes in $O(|V||E|)$ time.

The time complexity for repeating the search step once becomes $O(3*|V|+|V|\log|V|+2*|E|+|V||E|) \subseteq O(|V|\log|V|+|V||E|)$. The total time complexity for the search step is $O(|V|*\log|V|^2*K*L+|V|\log|V|*K*L*|V||E|)$. The partition step involves finding the strongly connected components and the topological order which can be done in $O(|V|+|E|)$. Merging the solution can be done in $O(|V|*L)$. The time complexity for the all three steps becomes $O(|V|*\log|V|^2*K*L+|V|\log|V|*K*L*|V||E|+|V|+|E|+|V|*L) \subseteq O(|V|*\log|V|^2*K*L+|V|\log|V|*K*L*|V||E|) \subseteq O(|V|^2\log|V|*K*L*|E|)$. □

3.6.1 The partition step

The first step involves finding subgraphs for all strongly connected components in G. This can be done in $O(|V|+|E|)$ time using Tarjan's strongly connected components algorithm [15]. To minimise the length of the strategy found in the search step, the subgraphs are sorted in descending order based on an estimate of the search number. The estimate used is Edvin's estimate (see definition 22).

```
1  PARTITION_STEP(G)
2      // Compute a DAG of strongly connected components
3      DAG strong_components = TARJAN(G)
4      // Create a list of all strongly connected components
5      strong_components = empty list
6      foreach strong_component in DAG
7          strong_components.add(strong_components)
8      // Sort the list based on EDVIN_ESTIMATE
9      sort_descending(strong_components)
10     return (DAG, strong_components)
```

3.6.2 The search step

Once the strongly connected components are found, we perform a search for a winning strategy for each of these components. The search utilises a method *SOLVE* which returns a winning strategy with k pursuers for the graph G or "FAILURE" if no such strategy exists. A binary search will find the search number according to *SEARCH* in $O(\log|V|)$ steps. A linear search from the lower bound l to the upper bound u uses at most $|V(G)|$ steps, but stops directly after *SOLVE* has returned a winning strategy. Both a binary search and a linear search can be tweaked in many different ways. A binary search could, for example, be made between l and Edvin's estimate e. If no winning strategy could be found, a binary search between e and u is executed. The sample below uses a linear strategy from l to u.

```
1 SEARCH_STEP(G, lower, upper)
2     // indegree[i] should contain indegree for vertex i
3     indegree = G.get_indegree
4     for k=lower to upper
5         strategy = SOLVE(G, k, k, copy(indegree), copy(indegree), new
      array, 0)
6         if (strategy != "FAILURE")
7             return strategy
```

SOLVE calculates the next state in a recursive manner until a winning strategy has been found or the maximum stack depth (MAX_DEPTH) has been exceeded. A state consists of an array with the number of free edges for each vertex. A *free edge* is an incoming edge which is not blocked. That means, the number of free edges for a vertex v, is the number of edges $u \rightarrow v$ such that u is not decontaminated. If the number of free edges for v is zero, v is considered decontaminated. A winning strategy is found when there are no free edges in G left.

Upon invocation, *SOLVE* checks whether the current state has been visited (row 17). This check should not be performed in an intermediary step, but only after all k pursuers have been placed. A state is considered visited if the set of decontaminated vertices has been encountered in any previous state. If the state is converted into a bitvector b with a zero on position p if $currentState[p] = 0$ and one otherwise, it would be possible to store the visited states in an array with bits where $visited[b] = 1$ if state b is visited. However, since the number of states grows exponentially with the number of vertices in G, this approach would not work if G is large. Another approach would be to use a large bloom filter. A bloom filter has $O(1)$ lookup and insertion with a small risk of a false positive (a state is considered visited although it is not).

If the state is not visited and the step is not intermediary, we check if a winning strategy is reachable from the current state. This is done by counting the number of contaminated vertices. If this number is less than or equal to k, we have a winning strategy by positioning pursuers at the contaminated vertices The strategy which lead to the winning state is collected bottom-up using backtracking (row 20-24).

If the state is neither visited nor winning, then we should check stack depth (length of current strategy). If the stack depth exceeds MAX_DEPTH, then we must admit failure and backtrack (row 26-27). If we fail to do so, we will end up exhausting the stack depth and the program will crash. This will happen even for small graphs since the length of the strategy also grows exponentially with the number of vertices in G. The length of an optimal strategy is usually between $|V(G)|$ to $2 * |V(G)|$, and we suggest MAX_DEPTH to be set to a value in this interval.

A pursuer is positioned at the next available vertex v given by the selector (row 35), whereby the current and next state is updated. The current state is updated by setting $currentState[v] = 0$ (row 42) and the next state is updated by decrementing the number of free edges for each neighbour to v to by one (row 43-44). If *SOLVE* has positioned k pursuers, it is the monk's turn to move. When this happens, the current state becomes the next state (row 57), and the next state is recalculated according to the following principle: each vertex v should have its number of free edges set to

its indegree reduced by the number of edges $u \to v$ where u is decontaminated in the current state (row 59-76).

The method SELECT (row 29) is implemented by the selector, see 3.6.5.

Here follows an example of how such an algorithm can be implemented. Psuedocode for Edvin's estimate is available in section 22.

```
1  /*
2     G -- a strongly connected component to decontaminate
3     static_pursuers -- The number of pursuers used in this strategy
4     dyn_pursuers -- The number of pursuers left to place
5     current_state -- The current state in this strategy
6     next_state -- The next state in this strategy
7     vertices -- The vertices decontaminated by pursuers at each day
8     depth -- Length of this strategy
9  */
10 SOLVE(G, static_pursuers, dyn_pursuers, current_state, next_state,
       vertices, depth) {
11     // Set flag "new_day" if this step is the first intermediary step
12     new_day = (static_pursuers = dyn_pursuers)
13     // Set flag "last_pursuer" if this step is the last intermediary
       step
14     last_pursuer = (dyn_pursuers = 1)
15
16     if new_day
17         if currentState is visited
18             return "FAILURE"
19     mark currentState as visited
20     if number of contaminated vertices <= staticPursuers
21         winning_strategy := empty strategy
22         // Add vertices to decontaimate at day "depth"
23         winning_strategy.add(contaminated vertices)
24         return strategy
25
26     if depth > MAX_DEPTH
27         return "FAILURE"
28
29     vertices_to_decontaminate = SELECT(G, current_state, next_state,
       static_pursuers, dyn_pursuers)
30
31     foreach vertex in vertices_to_decontaminate
32         if vertex is decontaminated
33             skip
34
35         vertices.add(vertex)
36
37         // Block edges originating from the current vertex
38         new_current_state = copy(current_state)
39         new_next_state = copy(next_state)
40
41         // Decontaminate vertex
42         new_currrent_state[vertex] = 0
43         foreach neighbour to vertex
```

```
            new_next_state[neighbour] = new_next_state[neighbour]-1

        if last_pursuer
            foreach vertex in G
                new_next_state[vertex] = indegree for vertex
            foreach decontaminated vertex v in new_current_state
                foreach neighbour to v
                    new_next_state[neighbour] = new_next_state[
    neighbour]-1

        strategy = SOLVE(
            G,
            static_pursuers,
            if last_pursuer then static_pursuers else dyn_pursuers-1,
            if last_pursuer then new_next_state else new_current_state,
            if last_pursuer then TRANSITION(G, new_next_state) else
    new_next_state,
            if last_pursuer then new array else vertices,
            if last_pursuer then depth + 1 else depth
        )
        if strategy != "FAILURE"
            if last_pursuer
                strategy.add(vertices)
                return strategy

        return "FAILURE"

TRANSITION(G, state)
    next_state = copy(state)
    foreach vertex in G
        next_state[vertex] = indegree for vertex
    foreach decontaminated vertex v in state
        foreach neighbour to v
            next_state[neighbour] = next_state[neighbour]-1
    return next_state
```

3.6.3 The merge step

The solutions should be merged in the order defined by the DAG of strongly connected components. This DAG can be built while searching for strongly connected components or in a separate step.

```
/*
    DAG -- a directed acyclic graph of strongly connected components
    stable_components -- the strongly connected components in the DAG
    strategies -- the strategy for each of the stable components,
    should appear in the same order as in stable_components
*/
MERGE_STEP(DAG, stable_components, strategies)
    strategy = empty list
    foreach strong_component of DAG in toplogical order
        strategy.add(strategies[strong_component])
    return strategy
```

3.6.4 Putting it all together

```
/*
    Find a strategy to decontaminate the graph G with a few
    pursuers as possible.
*/
SOLVE(G)
    (DAG, strong_components) = PARTITION_STEP(G)
    strategies = empty list
    // The maximum pursuers needed for solved components,
    // approximately the search number.
    int search_num_approx = 0
    foreach strong_component in strong_components in order
        lowerBound = lowerBound(G)
        upperBound = upperBound(G)
        strategy = SEARCH_STEP(
            G,
            max(lowerBound, search_num_approx),
            upperBound
        )
        search_num_approx = max(search_num_approx, pursuers(strategy))
        strategies.add(strategy)
    return MERGE_STEP(DAG, stable_components, strategies)
```

Before invoking the *SEARCH_STEP*, one could check if there exists a trivial solution (theorem 13). This is done by computing the intersection of the vertex sets for all elementary circuits.

3.6.5 The selector

The selector implements a method *SELECT* which takes a strongly connected component, the current and the next state, the number of pursuers used in the strategy, and the number of pursuers left to place. The method returns a list of vertices to decontaminate in the next state. The vertex to be decontaminated first should appear in the beginning of the list. If the selector returns more than one vertex, *SEARCH* will create branches, where each branch is inspected separately. If one branch does not yield a solution, the next branch will be tried. If the selector returns a list of all vertices, the search step becomes a *brute-force search*. Our implementation returns a single vertex based on figure 3.7.

Chapter 4

Discussion

It is still unknown whether there exist a polynomial time algorithm for finding an optimal strategy for the monk problem. If such an algorithm exists, it would imply that both MSNDP and MSLDP belong to P. If it is possible to bound the length of optimal solutions by a polynomial expression, then MSNDP and MSLDP are in NP because the implementation of EL-systems would run in polynomial time. An implication of conjecture 6 would be, that there exists a polynomial time reduction of a graph G in such a way that the search number for G becomes equal to the size of the maximum clique in G.

Open problem 1. *What is the relation between the search number and the size of the maximum clique in a monk graph?*

Regarding the cycle decomposition, it might not be feasible for practical applications. Even though finding all cycles in a graph can be done effectively by depth first traversal using Johnson's algorithm running in $O((n + e)(c + 1))$ where n is the number of vertices, e is the number of edges, and c is the number of elementary circuits [10], since the number of cycles can grow exponentially with $|G|$ it might not be practical to do cycle decomposition if G is large or dense.

An interesting property of the strategies we have found so far, is that the number of decontaminated vertices never decrease. Thus, we propose the following conjecture:

Conjecture 14. *Denote the number of decontaminated vertices in a monk graph G on day t with $D(G_t)$. For any optimal strategy of length $n+1$ to the monk problem, it holds that $D(G_i) \leq D(G_{i+1})$, $0 \leq i \leq n$.*

Since the monk problem can be seen a search game, it would be possible to implement the search step using the *minimax algorithm* with *alpha-beta pruning*. The number of decontaminated vertices could be used as analysis function. Such heuristics would be slower due to the number of branches which has to be inspected, opposed to the greedy selector where only one branch is followed.

The length of the optimal strategies l for the graphs solved was about $|V(G)| \leq l \leq 2 * |V(G)|$. No graph is known where the optimal strategy exceeds $2 * |V(G)|$. Increasing the number of pursuers decreased the length of the solution and the time it took to find it. Although we set an upper limit on $2 * |V(G)|$ in our heuristics to decrease the running time of the algorithm, we do not believe that the length of the optimal strategy is bounded by this constant. The question we ask is, what is the relation between the length of a solution and the number of pursuers?

Open problem 2. *What is the relation between the length and the number of pursuers in a winning strategy?*

We have noticed that the strategy produced with stable reduction (see lemma 10) is not necessarily optimal assuming all strategies of the stable components are optimal. We have considered a reduction of a monk graph G to G' which will implicitly cause the stable components to be solved in order, but a greedy algorithm could still focus on an optimal strategy for G'. This reduction works as follows:

1. Let $G' = G$.
2. Create a DAG D from the strongly connected components of G. Let $V(d)$ denote the vertices in the strongly connected component d.
3. If $u \to v \in E(D)$, then all edges from $V(u)$ will have edges to all vertices in $V(v)$ in G'.

However, the reduction may affect performance negatively since an algorithm would have to focus on the whole graph with added edges, and not parts of it. We believe that the divide and conquer approach is more suitable for large graphs, but the reduction approach is better for small graphs consisting of many stable components. This could be investigated in future research.

Chapter 5

Conclusion

An EL-system can be used as a verifier for MSNDP and MSLDP. Two implementations of an EL-system production process have been discussed, one using matrices and one operating on bitvectors.

The strongly connected components of a graph G can be decontaminated in sequence to form a winning strategy, and the search number is equal to the maximum search number of the strongly connected components in G.

Bounds for the monk problem have been established based on cycle decomposition, indegree and size of the maximum cliques. A polynomial heuristic approach have been proposed for finding near-optimal strategies.

Bibliography

[1] Aigner, M., and Fromme, M. A game of cops and robbers. *Discrete Applied Mathematics 8*, 1 (1984), 1 – 12.

[2] Britnell, J. R., and Wildon, M. Finding a princess in a palace: a pursuit-evasion problem. *Electr. J. Comb. 20*, 1 (2013), P25.

[3] Chou, H.-H., Ko, M.-T., Ho, C.-W., and Chen, G.-H. Node-searching problem on block graphs. *Discrete Applied Mathematics 156*, 1 (2008), 55 – 75.

[4] Clarke, N., Finbow, S., Fitzpatrick, S., Messinger, M., and Nowakowski, R. Seepage in directed acyclic graphs. *Australasian Journal of Combinatorics 43* (2009), 91–102.

[5] Dinneen, M. J. Vlsi layouts and dna physical mappings. Tech. rep., Citeseer, 1996.

[6] Fomin, F., Golovach, P., and Kratochvíl, J. On tractability of cops and robbers game. In *Fifth Ifip International Conference On Theoretical Computer Science – Tcs 2008*, G. Ausiello, J. Karhumäki, G. Mauri, and L. Ong, Eds., vol. 273 of *IFIP International Federation for Information Processing*. Springer US, 2008, pp. 171–185.

[7] Hunter, P., and Kreutzer, S. Digraph measures: Kelly decompositions, games, and orderings. *Theoretical Computer Science 399*, 3 (2008), 206 – 219. Graph Searching.

[8] Isaacs, R. *Differential games. A mathematical theory with applications to warfare and pursuit, control and optimization.* (The SIAM Series in Applied Mathematics) New York-London-Sydney: John Wiley and Sons, Inc. XXII, 384 p. , 1965.

[9] Isler, V., Kannan, S., and Khanna, S. Randomized pursuit-evasion with local visibility. *SIAM Journal on Discrete Mathematics 1*, 20 (2006), 26–41.

[10] Johnson, D. B. Finding all the elementary circuits of a directed graph. *SIAM J. Comput. 4*, 1 (1975), 77–84.

[11] Kirousis, L. M., and Papadimitriou, C. H. Searching and pebbling. *Theoretical Computer Science 47*, 0 (1986), 205 – 218.

[12] Meister, D., Telle, J. A., and Vatshelle, M. Recognizing digraphs of kelly-width 2. *Discrete Applied Mathematics 158*, 7 (2010), 741 – 746. Third Workshop on Graph Classes, Optimization, and Width Parameters Eugene, Oregon, USA, October 2007.

[13] Prusinkiewicz, P., and Lindenmayer, A. *The Algorithmic Beauty of Plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[14] Seymour, P. D., and Thomas, R. Graph searching and a min-max theorem for tree-width. *J. Comb. Theory Ser. B 58*, 1 (May 1993), 22–33.

[15] Tarjan, R. E. Depth-first search and linear graph algorithms. *SIAM J. Comput. 1*, 2 (1972), 146–160.