Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Cops and Robbers

Bachelor Thesis

Joël Rickert

rickertj@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
G. Bachmeier, S. Brandt
Prof. Dr. Roger Wattenhofer

Thursday 15th June, 2017

Abstract

This paper studies the game of cops and robbers. We present an algorithm to compute how long it takes the cops to capture the robber on a restricted graph class, by using information of the graph class. This restricted graph class is designed to allow the robber to evade capture for a long time. Then another algorithm is introduced that computes the time it takes the cops to capture the robber on any graph. Moreover it is shown how being able to compute the time it takes cops to capture the robber can be used to determine how many cops it takes to capture the robber. Both algorithms can be used to confirm results of a recent theoretical paper[5] computationally. The second algorithm is then used to calculate the number of cops it takes to capture a robber in a few interesting graphs. Furthermore we present an other graph class that requires many cops to capture the robber.

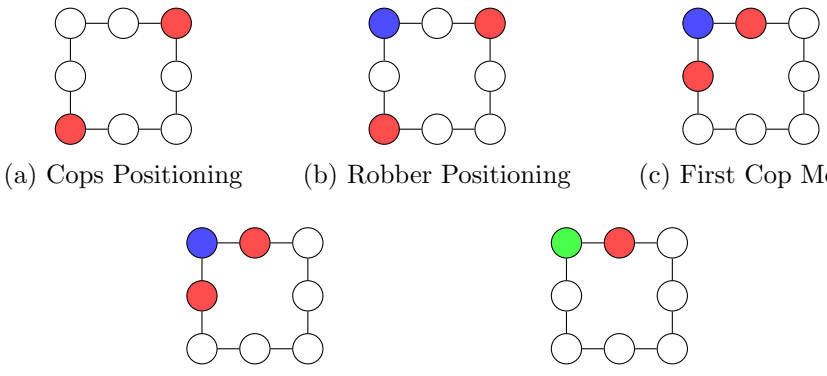
Contents

Chapter 1

Introduction

The game of cops and robbers is played on a connected graph. It is played by two players, a robber player and a cop player. The cop player has k cops he can move, whilst the robber player has only one robber. All the cops and the robber are positioned on nodes of the graph. The goal of the game is for the cops to reach the same node as the robber and capture him. The robber on the other hand tries to no get captured. Both the cops and the robber are able to see all the positions. The cop player coordinates the moves of all the cops.

The cops first choose their positions on the graph. When they have chosen all their positions the robber can choose his position. They now take turns moving starting with the cops. When a player moves he either stays on its node or moves to a neighbour node. Both players move until a cop reaches the same position as the robber, thus capturing the robber. If no cop ever reaches the same position as the robber, the cops did not capture the robber. An example of the game of cops and robbers is demonstrated in Figure 1.1, where the cops are coloured red and the robber is coloured blue. The node that is coloured green is where a cop captured the robber.



(a) Cops Positioning (b) Robber Positioning (c) First Cop Move

(d) First Robber Move (e) Cops Capture the Robber

Figure 1.1: First Cops and Robber Game

We study the capture time of graphs, which is the smallest time the cops can capture a robber that tries his best to not get captured. The capture time depends on the number of cops and has to be finite. If with a certain number of cops the robber can not be captured there is no capture time for this amount of cops. On a graph with n nodes and k cops, there are n^{k+1} possible ways to arrange all the cops and robbers. Therefore the capture time has to be in the order of $O(n^{k+1})$. A recent result [5] has shown a graph class that has also a capture time in the order of $\Omega(n^{k+1})$. We present an algorithm that computes how long it takes to capture the robber in this graph class, confirming this recent result. To faster compute this, knowledge of strategies for the cops and robbers is used. The runtime of the algorithm is $O\left(\Delta(G)n^{k+1}\right)$, where $\Delta(G)$ is the maximal amount of neighbours any node has in the graph G.

Then we develop a different algorithm that can be used for any graph. This algorithm has a runtime of $O\left(n\binom{n}{k}k\log k((\Delta(G))^k + (\Delta(G))^2)\right)$, making it slower than the first algorithm, but more versatile. We show how this algorithm is usable to determine, how many cops it takes to capture the robber on a graph. Furthermore the number of cops it takes to capture the robber is computed for some graphs along with the time it takes them to capture the robber.

Moreover a construction of another graph class is shown. These graphs require k cops to capture the robber in graphs with $k^2 + k$ nodes. Whilst this construction is not given explicitly for all the edges, with the insight of smaller constructed graphs, one might find a way to explicitly describe all the edges.

Chapter 2

The Game of Cops and Robbers

2.1 Definitions and Notations

The game of cops and robbers is played on a connected, simple and undirected graph $G(V, E)$. We call the number of nodes $n = |V|$ and denote the number of cops k. The game is also played with perfect information, therefore both parties know the positions of the other party. The goal of the cops is to capture the robber, while the robber tries to not get captured. To describe the positions of the cops and robber configurations are used.

Definition 2.1 (Configuration)**.** A *configuration conf* is a tuple $(r, c_0, \ldots, c_{k-1}) \in V^{k+1}$ where r is the position of the robber, and $c_0, \ldots, c_{k-1} \in V^k$ a *cop configuration cop_conf*. In a cop configuration $c_i \in V$ for $i \in \{0, \ldots, k-1\}$ is a position of a cop. When using configurations that change over time, then $conf(t)$ is $(r(t), c_0(t), \ldots, c_{k-1}(t))$ and $cop\_conf(t)$ its cop configuration.

Definition 2.2 (Neighbours)**.** We define $N^+(u) = \{v \in V \mid v = u \text{ or } \{u, v\} \in E\}$ for any node $u \in V$. The maximal number of neighbours a node has in the graph G is denoted $\Delta(G)$, while the minimal number of neighbours is denoted $\delta(G)$.

Definition 2.3 (Cover)**.** A node $u \in V$ is called covered by the cops, if there exists a $i \in \{1, \ldots, k-1\}$, such that $u \in N^+(c_i)$.

Now it is important what moves are allowed. If the robber makes a move from $conf(t-1)$ to $conf(t)$ it is called a *robber move*. To be a valid robber move, firstly $r(t) \in N^+(r(t-1))$ has to hold. The second thing that needs to hold is that $cop\_conf(t) = cop\_conf(t-1)$. When the cops make a move from $conf(t-1)$ to $conf(t)$ it is called a *cop move*. A valid cop move has to fulfill similar conditions. The first condition that has to uphold is that $c_i(t) \in N^+(c_i(t-1))$ for $i \in \{1, \ldots, k-1\}$. The second condition is that $r(t) = r(t-1)$. A sequence of configurations is also called a path p. To be a valid path the first move has

to be a cop move, and afterwards cop and robber moves alternate. The cops capture the robber on a path if there exists a $t < \infty$ with $r(t) = c_i(t)$ with $i \in \{1, \ldots, k-1\}$. The time it takes to capture the robber $t_{cap}(p)$ is first time the robber is captured on the path.

Definition 2.4 (Optimal Play)**.** The game of cops and robbers is played optimally on a path, if it is not possible for the cops to force capture sooner. Also it is not possible for the robber to evade capture for longer.

Definition 2.5 (Capture time)**.** The *capture time* of a graph $t_{cap}(G)$ is defined as $t_{cap}(G) = t_{cap}(p)$ where p is a path on G, which the game is played optimally on. Therefore the capture time has also to be finite.

Definition 2.6 (Cop number)**.** The smallest amount of cops k needed to force capture on a graph G is called the *cop number* $c(G)$ of the graph.

When the game of cops and robbers is played on a graph G with with $k < c(G)$ cops, it is possible for the robber to evade capture infinitely long. The cop number of a graph is always finite because when a graph has n nodes with n cops the robber is easily captured by putting one cop in each node.

2.2 Related Work

the theorem presented here is useful when trying to construct graphs with specific cop numbers, since when fulfilling all the criteria in the theorem, the constructed graph has lower bound to its cop number.

Theorem 2.7 (Aigner and Fromme, 1984. [1])**.** *Let G be a graph with minimum degree $\delta(G)$ which contains no 3- or 4-cycles. Then $c(G) \geq \delta(G)$.*

Proof Scetch Let G be a graph and $k = \delta(G) - 1$ the number of cops. Since each node in G has at least $\delta(G)$ many neighbours, the number of nodes in G has to be bigger than $\delta(G)$. Therefore for each cop configuration there exists at least one node u where no cop is positioned in. Now the robber gets put on u. Because there are no 3-cycles in G no two neighbours of u have an edge between them, and because there are no 4-cycles in G there is no node $v \in V$, with $v \neq u$, that has an edge to two neighbours of u. Therefore there is no node except for u that has more than one edge to a neighbour of u, and since u has at least $\delta(G)$ neighbours k cops can not cover all the neighbours of u. In conclusion $\delta(G) - 1$ cops are not enough to capture the robber, therefore the $c(G) \geq \delta(G)$.

Chapter 3

Maximal Capture Time Graphs

The graph $G(V, E)$ with k cops has n^{k+1} possible configurations the game of cops and robber can be in. When both parties play optimally, it can not be possible that the cops move into a configuration that was already played, and that the cops are able to capture the robber. This means if there are two configurations $conf(t_1)$ and $conf(t_2)$ with $conf(t_1) = conf(t_2)$ and $t_1 < t_2$, the cops could at time t_1 move to $conf(t_2 + 1)$ instead of $conf(t_1 + 1)$ and capturing the robber earlier. That shortcut is a contradiction to the optimality of the path to capture. Therefore is an optimal path of length smaller or equal to $2n^{k+1}$. Furthermore has the capture time of a graph to be smaller or equal to $2n^{k+1}$, hence being of the order of n^{k+1}. But how large could the capture time get? There is a recent result for this, which is the following theorem:

Theorem 3.1 (A Tight Lower Bound for the Capture Time of the Cops and Robbers Game[5])**.** *There exists a universal positive constant α such that for every $k \geq 2$, there exists an infinite family $\mathcal{G}$ of k-cop-win graphs such that the capture time of any n-vertex graph $G \in \mathcal{G}$ is at least $(n/(\alpha k))^{k+1}$. Moreover, the smallest graph in $\mathcal{G}$ has $n = O(k^2)$ nodes.*

If the number of cops is fixed, using this theorem, we get a family of graphs that have a lower bound for the capture time of $\Omega(n^{k+1})$. The trivial upper bound of $O(n^{k+1})$ was also shown before. Putting the two bounds together results in the capture time being of order $\Theta(n^{k+1})$.

3.1 Construction

Here we give an explicit construction of those graphs used in Theorem 3.1. Since there are many nodes and edges, first a list of all the nodes is given and afterwards the edges are listed. The construction can also be found, in more detail, in the paper[5] that gives us the theorem.

3.1.1 List of Nodes

For the construction of the graph we need a value $\hat{n}$, which scales linearly to n. The value $\hat{n}$ represents the size of all the components that grown for a fixed amount of cops. It is required that $\hat{n}$ is dividable by three. Then we take the number of cops and get the family of graphs $\left\{G^k_{\hat{n}}\right\}_{\hat{n}\geq k,\hat{n}\equiv 0(\text{mod } 3)}$. The vertices of the Graph $G^k_{\hat{n}}$ are:

$$V(G^k_{\hat{n}}) = \mathcal{E} \cup \mathcal{L}^* \cup \mathcal{S} \cup \mathcal{A} \cup \mathcal{T} \cup \mathcal{R} \cup \mathcal{C} \cup \mathcal{D}^0 \cup \mathcal{D}^1 \cup \ldots \cup \mathcal{D}^{k-2} \cup \{\mathcal{P}\} \cup \mathcal{X}.$$

$\mathcal{E}$, $\mathcal{L}^*$ and $\mathcal{X}$ are taken out of a graph we call $G^k_{\mathcal{E},\mathcal{L}}$. This bipartite graph is designed in a way that k cops can't capture the robber in it. To construct $G^k_{\mathcal{E},\mathcal{L}}$ we need a prime number p. The number p is chosen such that $2k + 10 \leq p$ and p is minimal. With this we construct the following two components

$$\mathcal{E} = \{\mathcal{E}_{i,j} \mid 0 \leq i < p \wedge 0 \leq j < p\} \text{ and}$$
$$\mathcal{L} = \{\mathcal{L}_{i,j} \mid 0 \leq i < p-1 \wedge 0 \leq j < p\}.$$

Using $G^k_{\mathcal{E},\mathcal{L}}$ we create the vertex sets $\mathcal{E}$, $\mathcal{L}^*$ and $\mathcal{X}$. The set $\mathcal{X}$ we call the exits of the graph, because these will be the nodes the robber will try to escape to, which in turn the cops will have to prevent, since it leads to the component they can't catch him in. Also we define $\mathcal{U} = \mathcal{E} \cup \mathcal{L}^*$ as this is the component the robber in the strategy will mainly try to reach. Then we take the nodes $\mathcal{E}$ from $G^k_{\mathcal{E},\mathcal{L}}$ as they are. But for the components $\mathcal{L}^*$ and $\mathcal{X}$ we take the following nodes out of $G^k_{\mathcal{E},\mathcal{L}}$:

$$\mathcal{L}^* = \{\mathcal{L}_{i,j} \mid 0 \leq i < 2k \wedge 0 \leq j < p\},$$
$$\mathcal{X} = \mathcal{X}^0 \cup \mathcal{X}^1 \cup \mathcal{X}^2 \text{ where}$$
$$\mathcal{X}^0 = \left\{\mathcal{X}^0_0, \ldots, \mathcal{X}^0_{6(k-1)-1}\right\} \subseteq \{\mathcal{L}_{h,j} \mid 2k + j * 3 \leq h \leq 2k + 2 + 3j \wedge 0 \leq j < p\},$$
$$\mathcal{X}^1 = \left\{\mathcal{X}^1_0, \ldots, \mathcal{X}^1_{6(k-1)-1}\right\} \subseteq \{\mathcal{L}_{h,j} \mid 2k + 3 \leq h \leq 2k + 5 \wedge 0 \leq j < p\} \text{ and}$$
$$\mathcal{X}^2 = \left\{\mathcal{X}^2_0, \ldots, \mathcal{X}^2_{6(k-1)-1}\right\} \subseteq \{\mathcal{L}_{h,j} \mid 2k + 6 \leq h \leq 2k + 8 \wedge 0 \leq j < p\}.$$

The next nodes are $\mathcal{S}$ these are the nodes the cops can stand in to cover all the nodes of $\mathcal{E}$, $\mathcal{L}^*$ and $\mathcal{X}$. They are designed to be the only way to force the robber out of that part of the graph. This is why in an optimal game the cops will start in this component. $\mathcal{S}$ is just a node for every cop, hence $\mathcal{S} = \{\mathcal{S}_0, \ldots, \mathcal{S}_{k-1}\}$. The two components $\mathcal{A}$ and $\mathcal{T}$ are designed to connect other components in a way that the cops can't abuse components to capture the robber earlier and to prevent the robber from escaping in other ways than we want. The number of nodes they each have is two times the number of cops. This leads to $\mathcal{A} = \{\mathcal{A}_0, \ldots, \mathcal{A}_{2k-1}\}$

and $\mathcal{T} = \{\mathcal{T}_0, \ldots, \mathcal{T}_{2k-1}\}$. The component $\{\mathcal{P}\}$ serves a similar purpose but it is just one node.

Furthermore is a list of the nodes that achieve the long capture time on this graph. The first component to achieve that is the $\mathcal{R}$ component. This is the component where the robber gets forced around. In an ideal game the robber will start in this component. $\mathcal{R}$ will be connected with the exits, so the cops will have to pay attention to cover all those at all times. The robber can be caught at either end of the component. We have that $\mathcal{R} = \{\mathcal{R}_{-\hat{n}}, \ldots, \mathcal{R}_{\hat{n}}\}$. The components $\mathcal{C}, \mathcal{D}^0, \mathcal{D}^1, \ldots$ and $\mathcal{D}^{k-2}$ are all cycles with the length $\hat{n}$. In each one of them there is a cop. They are connected with the exits in a way such that if the i-th cop wants to move one forward in its component the cop $i+1$ has to move forward a third of his component. And since this holds true for all the cops, the cop in the $\mathcal{C}$ component rarely moves, this is the component that is connected to the one the robber walks in. This gives us:

$$\begin{aligned} \mathcal{C} &= \{\mathcal{C}_0, \ldots, \mathcal{C}_{\hat{n}-1}\} \text{ and} \\ \mathcal{D}^j &= \left\{\mathcal{D}^j_0, \ldots, \mathcal{D}^j_{\hat{n}-1}\right\} \text{ for all } 0 \leq j < k-1. \end{aligned}$$

When we count together the number of edges we get:

$$\begin{aligned} p^2 + 2kp + k + 2k + 2k + 2\hat{n} - 1 + \hat{n} + (k-1)\hat{n} + 1 + 3 \cdot 6(k-1) = \\ p^2 + 2kp + 23k + (k+2)\hat{n} - 18. \end{aligned}$$

When we consider the fact that between each number and twice that number there is at least one prime number. We get that there are $O(k^2 + k\hat{n})$ many nodes in the graph.

3.1.2 List of Edges

The edges in $\mathcal{E}$, $\mathcal{L}^*$ and $\mathcal{X}$ are taken from the graph $G^k_{\mathcal{E},\mathcal{L}}$. The edges in the graph $G^k_{\mathcal{E},\mathcal{L}}$ are constructed in the following way, $\mathcal{L}_{i,j}$ is connected to all the nodes in $\left\{\mathcal{E}_{h,h(i+1)+j \pmod p} \mid 0 \leq h < p\right\}$. But these nodes in our graph are also connected to other components. For $\mathcal{E}_{i,j}$ those nodes are:

$$\begin{aligned} &\mathcal{S}_i \pmod k, \\ &\mathcal{A}_j \pmod{2k} \text{ and} \\ &\mathcal{T}_j \pmod{2k} \text{ if } i \pmod k = 1. \end{aligned}$$

On the other hand a node $\mathcal{L}_{i,j}$ is connected to the following nodes:

$$\begin{aligned} &\mathcal{S}_i \pmod k, \\ &\mathcal{A}_j \pmod{2k} \text{ and} \\ &\mathcal{T}_i \pmod{2k}. \end{aligned}$$

The additional edges with which the $\mathcal{X}$ component is connected we only describe from the perspective of the other node connected to it, because there are connections to many components and they are not easy to describe out of the perspective of the component $\mathcal{X}$.

The next component is the $\mathcal{S}$ component. This component has three cases, the first are the edges to $\mathcal{S}_0$:

$$\begin{aligned}&\mathcal{C}_i \text{ for all } i,\\ &\mathcal{X}_i^j \text{ where } j \in \{0,1,2\} \text{ and } i \in \{0,1,2\},\\ &\text{every } \mathcal{E}_{i,j} \text{ with } i \pmod k = 0 \text{ and}\\ &\text{every } \mathcal{L}_{i,j} \text{ with } i \pmod k = 0.\end{aligned}$$

And $\mathcal{S}_1$ is connected to the following edges:

$$\begin{aligned}&\mathcal{T}_i \text{ for all } i,\\ &\mathcal{X}_i^j \text{ where } j \in \{0,1,2\} \text{ and } i \in \{3,4,5\},\\ &\mathcal{X}_i^j \text{ where } j \in \{0,1,2\} \text{ and } i \in \{6,7,8\} \text{ if } k \geq 3,\\ &\text{every } \mathcal{E}_{i,j} \text{ with } i \pmod k = 1 \text{ and}\\ &\text{every } \mathcal{L}_{i,j} \text{ with } i \pmod k = 1.\end{aligned}$$

And at last the most general case for all the other nodes in $\mathcal{S}$, namely the nodes $\mathcal{S}_{i\geq 2}$ are connected to the following nodes:

$$\begin{aligned}&\mathcal{D}_j^i \text{ for all } j,\\ &\mathcal{X}_h^j \text{ where } j \in \{0,1,2\} \text{ and } h \in \{6(i-1)+3, 6(i-1)+4, 6(i-1)+5\},\\ &\mathcal{X}_h^j \text{ where } j \in \{0,1,2\} \text{ and } h \in \{6i, 6i+1, 6i+2\} \text{ for all } i < k-1,\\ &\text{every } \mathcal{E}_{h,j} \text{ with } h \pmod k = i \text{ and}\\ &\text{every } \mathcal{L}_{h,j} \text{ with } h \pmod k = i.\end{aligned}$$

Now we list the neighbours of the components $\mathcal{A}, \mathcal{T}$ and $\{\mathcal{P}\}$. So here are the nodes $\mathcal{A}_i$ is connected to:

$$\begin{aligned}&\mathcal{T}_j \text{ for all } j,\\ &\mathcal{R}_0,\\ &\mathcal{D}_j^0 \text{ for all } j,\\ &\mathcal{E}_{j,h} \text{ for all } h \pmod{2k} = i \text{ and}\\ &\mathcal{L}_{j,h} \text{ for all } j \pmod{2k} = i.\end{aligned}$$

Where as $\mathcal{T}_i$ is connected to:

$\mathcal{S}_1$,
$\mathcal{P}$,
$\mathcal{A}_j$ for all j,
$\mathcal{T}_j$ for all $j \neq i$,
$\mathcal{D}_j^0$ for all j,
$\mathcal{X}_h^j$ where $j \in \{0,1,2\}$ and $h \in \{3,4,5\}$,
$\mathcal{X}_h^j$ where $j \in \{0,1,2\}$ and $h \in \{6,7,8\}$ if $k \geq 3$,
$\mathcal{E}_{j,h}$ for all $j \pmod k = i \wedge h \pmod{2k} = i$ and
$\mathcal{L}_{j,h}$ for all $j \pmod{2k} = i$.

And finally node $\mathcal{P}$ is connected to:

$\mathcal{T}_i$ for all i,
$\mathcal{R}_i$ for all i,
$\mathcal{D}_i^0$ for all i and
$\mathcal{X}_i^j$ where $j \in \{0,1,2\}$ and $i = 3$.

The next component we look at is the component the robber will move in. This component is internally connected as a line and to $\mathcal{C}$ in a way that the cop there can force him to move forward. More precisely the node $\mathcal{R}_i$ is connected to the following nodes:

$\mathcal{A}_j$ for all j if $i = 0$,
$\mathcal{R}_j$ for $j = i - 1$ and $j = i + 1$,
$\mathcal{C}_j$ for all $0 \leq j \leq \hat{n}/3$ if $i \pmod 3 = 0$,
$\mathcal{C}_j$ for all $\hat{n}/3 \leq j \leq 2\hat{n}/3$ if $i \pmod 3 = 1$,
$\mathcal{C}_0$ and $\mathcal{C}_j$ for all $2\hat{n}/3 \leq j < \hat{n}$ if $i \pmod 3 = 2$,
$\mathcal{P}$ and
$\mathcal{X}_h^j$ where $j = i \pmod 3$ and $h \in \{0, \ldots, 6(k-1)-1\}$.

In the component $\mathcal{C}$ the node $\mathcal{C}_i$ is connected to:

$\mathcal{S}_0$,
$\mathcal{R}_j$ for all $j \pmod 3 = 0$ if $0 \leq i \leq \hat{n}/3$,
$\mathcal{R}_j$ for all $j \pmod 3 = 1$ if $\hat{n}/3 \leq i \leq 2\hat{n}/3$,
$\mathcal{R}_j$ for all $j \pmod 3 = 2$ if $2\hat{n}/3 \leq i < \hat{n}$ or $i = 0$,
$\mathcal{C}_j$ for $j \equiv i - 1 \pmod{\hat{n}}$ and $j \equiv i + 1 \pmod{\hat{n}}$,
$\mathcal{X}_h^j$ where $j \in \{0, 1, 2\}$ and $h \in \{0, 1, 3\}$ if $i \pmod 3 = 0$,
$\mathcal{X}_h^j$ where $j \in \{0, 1, 2\}$ and $h \in \{0, 2, 3\}$ if $i \pmod 3 = 1$ and
$\mathcal{X}_h^j$ where $j \in \{0, 1, 2\}$ and $h \in \{1, 2, 3\}$ if $i \pmod 3 = 2$.

The last component is $\mathcal{D}$ and it split into two parts to describe its edges. Firstly come the edges of $\mathcal{D}_i^0$ to these nodes:

$\mathcal{A}_j$ for all j,
$\mathcal{T}_j$ for all j,
$\mathcal{D}_j^0$ for $j \equiv i - 1 \pmod{\hat{n}}$ and $j \equiv i + 1 \pmod{\hat{n}}$,
$\mathcal{P}$,
$\mathcal{X}_h^j$ where $j \in \{0, 1, 2\}$ and $h \in \{2, 4, 5\}$ if $0 \leq i < \hat{n}/3$,
$\mathcal{X}_h^j$ where $j \in \{0, 1, 2\}$ and $h \in \{2, 4, 5\}$ if $\hat{n}/3 \leq i < 2\hat{n}/3$ and
$\mathcal{X}_h^j$ where $j \in \{0, 1, 2\}$ and $h \in \{0, 4, 5\}$ if $2\hat{n}/3 \leq i < \hat{n}$.
Additionally, if $k \geq 3$:
  $\mathcal{X}_h^j$ where $j \in \{0, 1, 2\}$ and $h \in \{6, 7, 9\}$ if $i \pmod 3 = 0$,
  $\mathcal{X}_h^j$ where $j \in \{0, 1, 2\}$ and $h \in \{6, 8, 9\}$ if $i \pmod 3 = 1$ and
  $\mathcal{X}_h^j$ where $j \in \{0, 1, 2\}$ and $h \in \{7, 8, 9\}$ if $i \pmod 3 = 2$.

Second are the edges from $\mathcal{D}_i^j$ to the following nodes:

$\mathcal{S}_{j+1}$
$\mathcal{D}_h^j$ for $h \equiv i - 1 \pmod{\hat{n}}$ and $h \equiv i + 1 \pmod{\hat{n}}$,
$\mathcal{X}_h^q$ where $q \in \{0, 1, 2\}$ and $h \in \{6j + 2, 6j + 4, 6j + 5\}$ if $0 \leq i < \hat{n}/3$,
$\mathcal{X}_h^q$ where $q \in \{0, 1, 2\}$ and $h \in \{6j + 1, 6j + 4, 6j + 5\}$ if $\hat{n}/3 \leq i < 2\hat{n}/3$ and
$\mathcal{X}_h^q$ where $q \in \{0, 1, 2\}$ and $h \in \{6j + 0, 6j + 4, 6j + 5\}$ if $2\hat{n}/3 \leq i < \hat{n}$.
Additionally, if $k \geq 3$:
  $\mathcal{X}_h^q$ where $q \in \{0, 1, 2\}$ and $h \in \{6j + 6, (j + 7, 6j + 9\}$ if $i \pmod 3 = 0$,
  $\mathcal{X}_h^q$ where $q \in \{0, 1, 2\}$ and $h \in \{6j + 6, 6j + 8, 6j + 9\}$ if $i \pmod 3 = 1$ and
  $\mathcal{X}_h^q$ where $q \in \{0, 1, 2\}$ and $h \in \{6j + 76j + 8, 6j + 9\}$ if $i \pmod 3 = 2$.

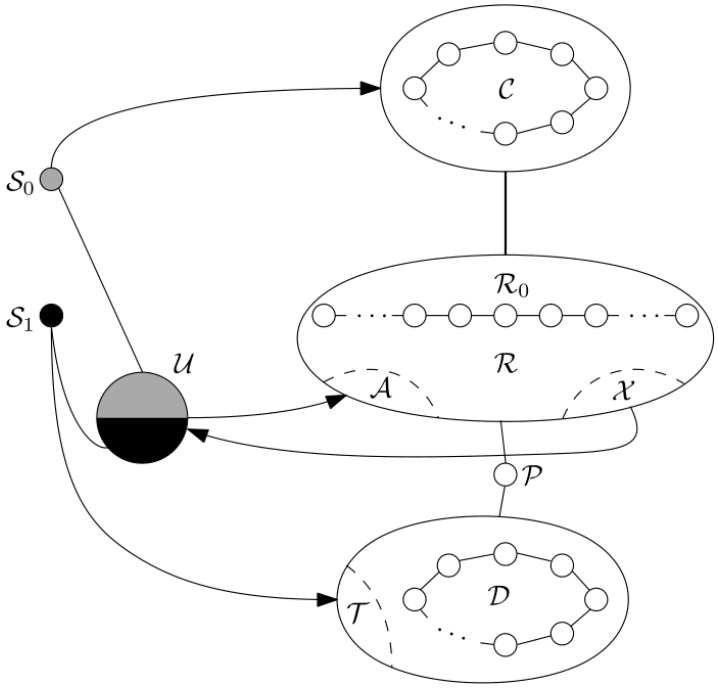


Figure 3.1: Components and Connections[5]

When looking at the connections of the components of the graph it will look like Figure 3.1.

3.1.3 Strategies

To capture the robber as fast as possible, the cops try to prevent the robber from getting to an exit node or into the $\mathcal{U}$ component of the graph. This means while the robber is in $\mathcal{R}$ they need to make sure that all the exits the robber can reach are covered, therefore the cops start in $\mathcal{S}_0, \ldots, \mathcal{S}_{k-1}$. The robber on the other hand tries to get to $\mathcal{U}$, so if not all of it is covered he would choose some uncovered node in $\mathcal{U}$ and start there. If the entire $\mathcal{U}$ is covered, then the robber will start in $\mathcal{R}_0$. If the cops can capture the robber, in one move, they will do so, otherwise they will move according to the Table 3.1. Meanwhile the robber will move according to Table 3.2.

These strategies are not the optimal way both players can play. How good they are can be seen in [5]. They are only a linear factor worse than the optimal strategy. One example how to make the cops strategy better is instead of pushing the robber only to one end of $\mathcal{R}$, they could push him onto the end that the robber is nearer to.

Robber Position	Cop Configuration	Next Cop Configuration
$\mathcal{A}_i$	$(\mathcal{S}_0, \ldots, \mathcal{S}_{k-1})$	$(\mathcal{S}_0, \mathcal{T}_i, \mathcal{S}_2, \ldots, \mathcal{S}_{k-1})$
$\neq \mathcal{A}_i$ for all i	$(\mathcal{S}_0, \ldots, \mathcal{S}_{k-1})$	$(\mathcal{S}_0, \mathcal{T}_j, \mathcal{S}_2, \ldots, \mathcal{S}_{k-1})$ for some j
arbitrary	$(\mathcal{S}_0, \mathcal{T}_i, \mathcal{S}_2, \ldots, \mathcal{S}_{k-1})$	$\left(\mathcal{C}_0, \mathcal{D}_0^0, \ldots, \mathcal{D}_0^{k-2}\right)$
$\neq \mathcal{C}_h$ for all h	$\left(\mathcal{C}_{j_0}, \mathcal{D}_{j_1}^0, \ldots, \mathcal{D}_{j_{k-1}}^{k-2}\right)$	the first of the following cop configurations that covers all exits: $\left(\mathcal{C}_{j_0+1}, \mathcal{D}_{j_1+1}^0, \ldots, \mathcal{D}_{j_{k-1}+1}^{k-2}\right)$ $\left(\mathcal{C}_{j_0}, \mathcal{D}_{j_1+1}^0, \ldots, \mathcal{D}_{j_{k-1}+1}^{k-2}\right)$ $\left(\mathcal{C}_{j_0}, \mathcal{D}_{j_1}^0, \mathcal{D}_{j_2+1}^0, \ldots, \mathcal{D}_{j_{k-1}+1}^{k-2}\right)$ $\vdots$ $\left(\mathcal{C}_{j_0}, \mathcal{D}_{j_1}^0, \ldots, \mathcal{D}_{j_{k-1}+1}^{k-2}\right)$ where all the indices are (mod $\hat{n}$)
$\mathcal{C}_h$	$\left(\mathcal{C}_{j_0}, \mathcal{D}_{j_1}^0, \ldots, \mathcal{D}_{j_k-1}^{k-2}\right)$	$(\mathcal{S}_0, \mathcal{P}$, arbitrary, …, arbitrary)

Table 3.1: Cop Strategy

Robber Position	Cop Configuration	Next Robber Position
some node in $\mathcal{U}$	$\neq (\mathcal{S}_0, \ldots, \mathcal{S}_{k-1})$	some uncovered node in $\mathcal{U}$
some node in $\mathcal{U}$	$(\mathcal{S}_0, \ldots, \mathcal{S}_{k-1})$	some node in $\mathcal{A}$
$\mathcal{A}_i$	$(\mathcal{S}_0, \ldots, \mathcal{S}_{k-1})$ or $(\mathcal{S}_0, \mathcal{T}_i, \mathcal{S}_2, \ldots, \mathcal{S}_{k-1})$	$\mathcal{R}_0$
$\mathcal{A}_i$	$\neq (\mathcal{S}_0, \ldots, \mathcal{S}_{k-1})$ and $\neq (\mathcal{S}_0, \mathcal{T}_i, \mathcal{S}_2, \ldots, \mathcal{S}_{k-1})$	some uncovered node in $\mathcal{U}$
$\mathcal{R}_i$	not covering all exits of $\mathcal{R}_i$	some uncovered exit of $\mathcal{R}_i$
$\mathcal{R}_i$	covering all exits of $\mathcal{R}_i$	the uncovered node from $\{\mathcal{R}_{i-1}, \mathcal{R}_i, \mathcal{R}_{i+1}\}$ with the smallest absolute index; if all are covered, stay in $\mathcal{R}_i$
$\mathcal{X}_i^j$	$\neq (\mathcal{S}_0, \ldots, \mathcal{S}_{k-1})$	some uncovered node in $\mathcal{U}$
$\mathcal{X}_i^j$	$(\mathcal{S}_0, \ldots, \mathcal{S}_{k-1})$	some uncovered node from $\{\mathcal{R}_{-1}, \mathcal{R}_0, \mathcal{R}_1\}$

Table 3.2: Robber Strategy

3.2 Algorithm

To test what the robbers could improve in their strategy, we try to figure out what moves the robber could make instead of his strategy to improve his play. To do this we use Algorithm 3.1 and Algorithm 3.2. Where the latter is based on a depth first search of a graph. The basic idea is that when the algorithm is at a configuration, it first does a cop moved based on the strategy of the cops. Furthermore the algorithm will try out all possible moves the robber can make, and pick the best one.

There are two maps used in these algorithms. The map `visited` is mapping configurations to booleans. If a configuration is not inserted into the map, it is equal to having the value `false`. This map is used to store all the configurations that the recursive algorithm has visited on its path. The other map `dist_to_capture` is mapping configurations to integers. It will represent the number of moves from the configurations until the robber is captured. The value `maximum` is the length of the longest path we have found for the robber to get captured. To figure out this value we run the recursive algorithm Algorithm 3.2 for every position the robber can choose to start in. To get the starting configurations, we always take the same cop configuration. The cops will start in $(\mathcal{S}_0, \ldots, \mathcal{S}_{k-1})$ according to their strategy. And to this we add every possible node on the graph as the starting position of the robber, and figure out for each one how long it takes the cops to capture him. If the algorithm returns ∞ it is not possible for the cops to capture the robber.

Algorithm 3.1 Algorithm to Compute Capture Time on Graph Family

```
1:  function RESTRICTED_ALGORITHM(Graph from family: G)
2:      init global: visited                    ▷ map: configuration → boolean
3:      init global: dist_to_capture            ▷ map: configuration → integer
4:      init: maximum ← 0                                          ▷ integer
5:      for all nodes u ≠ a position of a cop do
6:          init: conf = (conf.robber, conf.cops)                  ▷ configuration
7:          conf.cops ← (S_0, ..., S_{k-1})
8:          conf.robber ← u
9:          local_value ← REC_ALGORITHM(conf)                      ▷ Algorithm 3.2
10:         maximum ← max{maximum, local_value}
11:     end for
12:     return maximum
13: end function
```

The goal of Algorithm 3.2 is to recursively determine how long it takes the cops to capture the robber, starting from the configuration `conf`. First we check if on the path to `conf` we already passed `conf` because if this is the case the robber has the possibility to force an infinite loop, and can therefore not be

caught. To represent this we return the value ∞. After we have done this we check if we already know how long it takes to get captured. If we know, we return that value. Then we set the `visited` to `true` at `conf`. Then we set `conf'` to be the configuration after the cops make their move according to their strategy Table 3.1. If they capture the robber we update the `dist_to_capture` map with the values we figured out. Furthermore the algorithm returns one, since one turn after `conf` the robber gets captured. The last thing that needs to be done before the loop gets started is to prepare a value where we can save the best result of all the configurations the robber could chose for a next move.

The loop iterates over all the configurations `next_conf`, the robber could move to from `conf'`. For each configuration we call the recursive function again. Take the maximal value and add two to it, since there is a cop and a robber move more than in the configuration the recursive algorithm was called with. If there is a `next_conf` where the robber can avoid capture, we will just return ∞ since this is the optimal choice of the next step for the robber. The value that we calculated to be the most amount of moves the robber can escape capture will be stored in `dist_to_capture` and after that returned. But anywhere before there is a value returned, the value in `visited` gets set to false again at `conf`.

We can use the results of previous calls of the recursive function in the map `dist_to_capture` because the cops will always make the same move, and if there was a possibility of a loop on the path to capture, the algorithm would have figured that out before and would not have saved a finite value. This is the case because we go through the graph like a depth first search algorithm and therefore would reach any possible loop.

Runtime For a particular configuration the recursion can be called at most the number of possible ways a robber can reach said configuration. The number of configurations the loop runs through, is equal to the number of neighbours the node with the robber has plus one. And for each node the loop is run at most once, because if it is run once, the value `dist_to_capture` gets for said configuration in Algorithm 3.2 Line 25. Furthermore if the recursive algorithm is called with for a configuration that has a value in `dist_to_capture`, it will not go through the loop, but return a value earlier. Therefore the loop is at most run once for each configuration. If one takes into account, and that there are n^{k+1} configurations, the statements in the loop get executed at most $(\Delta(G)+1)n^{k+1}$ times. And because the recursion can be called at most once for each neighbour of a node, the recursion is called at most $(\Delta(G)+1)n^{k+1}$ times. This all together gets us an upper bound of the run time at $O(\Delta(G)n^{k+1})$. This makes it difficult to make computations of graphs with many cops. But since many configurations will never be reached, for example the cops will never go into the component $\mathcal{U}$. Hence the constant to this bound will at least be small, making it possible to compute the result for a good amount of different graphs in this family.

Algorithm 3.2 Recursive Algorithm

```
1: function Rec_Algorithm(Configuration: conf)
2:     if visited[conf] then
3:         return ∞
4:     end if
5:     if dist_to_capture[conf] is mapped then
6:         return dist_to_capture[conf]
7:     end if
8:     conf' ← next configuration according to cop strategy
9:     if captured then
10:        dist_to_capture[conf'] ← 0
11:        dist_to_capture[conf] ← 1
12:        return 1
13:    end if
14:    init: max_dist ← 0                                          ▷ integer
15:    for all configurations next_conf the robber can move to from conf' do
16:        visited[next_conf] ← true
17:        local_dist ← Rec_Algorithm(next_conf)
18:        visited[next_conf] ← false
19:        if local_dist = ∞ then
20:            dist_to_capture[conf] ← ∞
21:            return ∞
22:        end if
23:        max_dist ← max{max_dist, local_dist + 2}
24:    end for
25:    dist_to_capture[conf] ← max_dist
26:    return max_dist
27: end function
```

Space The space the maps could occupy in the worst case is in both cases n^{k+1}, with the same definitions of n and k as when calculating the run time. Since a value could be stored for each configuration. But the same as stated in considerations on the run time is true for the space, in that for many configurations the algorithm will never store any values. This is important when choosing data structures to implement this algorithm.

On the Implementation The algorithm was implemented in C++. But rather than using recursion, it was implemented iteratively. This is done, because for bigger inputs, the call stack would overflow and crash the algorithm. To omit the recursion, one must implement his own stack, to replace the call stack and loop over the manually implemented stack. Therefore the algorithm gets much more confusing, hence the algorithm is presented here in recursive form.

Chapter 4

The General Algorithm

In the previous chapter Algorithm 3.1 describes a way to compute the time it takes to capture the robber for a set cop strategy on a specific family of graphs. In this chapter we want to develop another algorithm that can figure out the capture time of any graph. One to compute the capture time, would be with a minimax algorithm. Then one would start in a configuration and build a tree with the next possible reachable configurations. Each new layer is either reached via a cop or a robber move. But even if α-β pruning [6] is used, it is very slow. So the idea that is used here computes how long it takes to get captured from all configurations at the same time, assuming both the cops and the robber play optimally.

We start with a graph G we want to compute the capture time of. To illustrate how the algorithm works, the algorithm is used on an example graph. The graph (Figure 4.1) used for the example are three nodes in a line with one cop. To compute the time it takes to get captured from each configuration, we build a meta graph G'. The nodes of G' are all the configurations of G. The nodes u and v in G' are connected with an edge, if there is either a cop or a robber move from u to v. For each configuration we want to compute how long it takes until the robber is captured, but since it makes a difference whether it is the cops or the robbers turn, the algorithm will calculate two values for each configuration. The time it takes to capture the robber if it is the cops turn to move is called the cop depth. Similarly the robber depth of a configuration is the time it takes for the robber to get captured if it is his turn. Initially we set all these depth values to ∞. Furthermore we set the values of the only configurations we initially know the value of, the configurations where the robber is captured. The configurations where the robber is captured have a robber and cop depth of zero. Figure 4.2 shows this initialisation done one the example graph. In the example representation of G' edges that correspond to cop moves are coloured red, while ones representing robber moves are coloured blue.

After all the nodes of G' are assigned to their initial cop and robber depth values, the goal of the algorithm is to assign depth values to all configurations

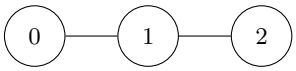


Figure 4.1: The graph

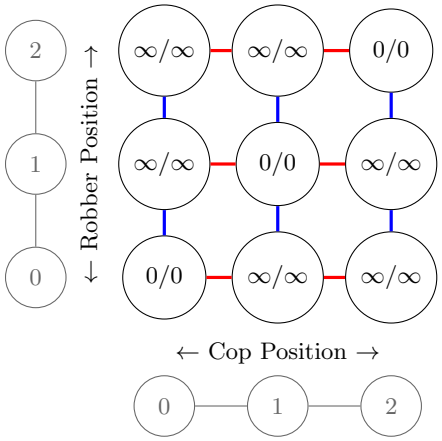
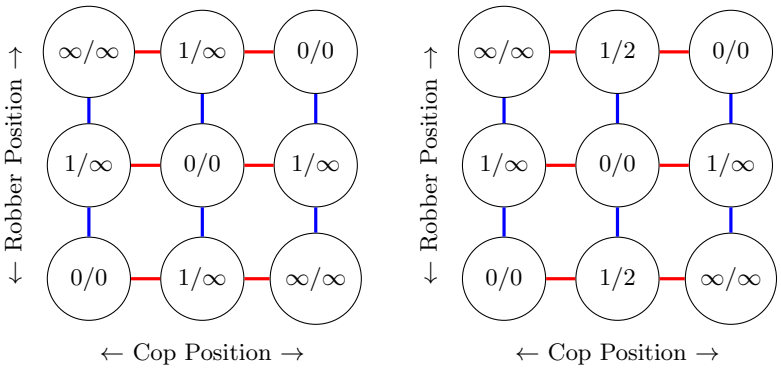


Figure 4.2: Initialization

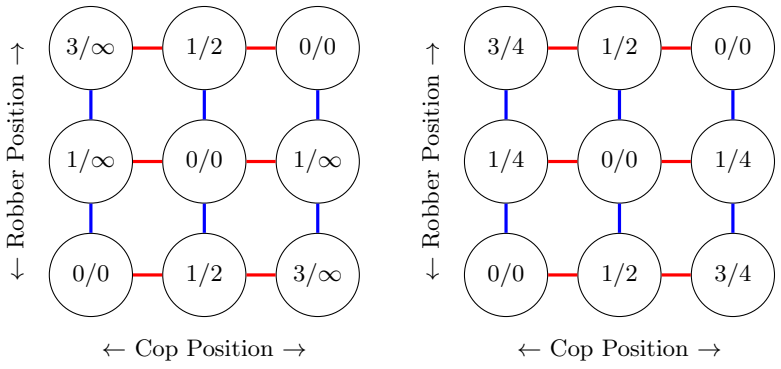
that are finite. A new cop depth is possible to compute if there is a node u with an infinite cop depth but a neighbour v that is reachable with a cop move that has a robber depth that is finite. The cop depth of u is then simply the robber depth of v plus one, if there are multiple way to chose v, the one chosen is the one with the smallest robber depth. To figure out the robber depth of a node u, all the neighbours v of u need to have a finite cop depth. Because if one has no finite cop depth, the best next move of the robber would be to move into that configuration. When all v have a finite cop depth, the robber depth of u will be the maximum of the cop depth values plus one. The algorithm will computed all the cop depth values it can, then it will compute all the robber depth values it can. This process is repeated until one of two things happen. Either the depth values of all the configuration are finite or there are no more depth values that are computable. In the example this is represented in Figure 4.3, and it takes four steps to fill in all the values.

With all assignable depth values assigned, it is time to determine what the actual capture time is. To do this it helps to remember how the game of cops and robbers begins. The game start by first the cops choosing their positions, afterwards the robber chooses his position. When all the positions are chosen it is the cops turn to make the first move. Therefore when choosing an optimal starting position, one must consider the cop depth. Furthermore for any cop configuration the cops start with, the robber will choose the position that creates the configuration with the highest cop depth. Therefore the cops chose their starting cop configuration such that this is minimized. With this in mind the algorithm will compute m_u to be the maximal cop depth, the robber could choose



(a) First cop depth step

(b) First robber depth step



(c) Second cop depth step

(d) Second robber depth step

Figure 4.3: All the steps

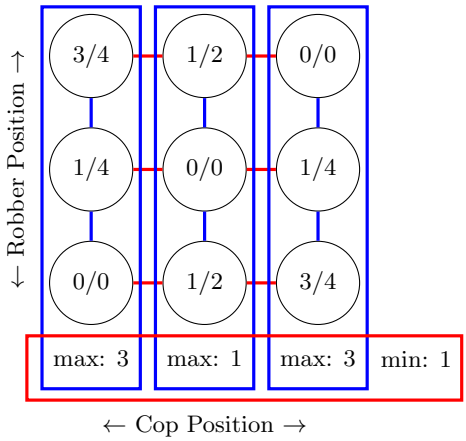


Figure 4.4: Finding the capture time

if the cops start in cop configuration u. Then the capture time the algorithm calculates is $\min m_u$ for all cop configurations u. Figure 4.4 illustrates this for the example. It is possible that the capture time the algorithm computes is infinite. When the computed capture time is infinite then the robber can avoid being captured.

4.1 The Algorithm in Detail

Data Structures Algorithm 4.1 is the algorithm that will hold all the subroutines that are described later. To store the cop and robber depth for each configuration two maps are used. The operations needed on maps is accessing them for an element, and a option to figure out whether an element is mapped or not. To keep track of the configuration the algorithm is calculating on, queues are used. The queues will need to support the multiple functions, the first one is push, which puts an element into the queue, where it is inserted is of no relevance. The second one is empty, which is true when there are no elements in the queue. The last operations are pop, which removes the first element in the queue, and front, which accesses the first element in the queue. The integer `i` is used to keep track on how many times the subroutines are called.

Initialization In Algorithm 4.2 all the values in `robber_depth` and `cop_depth` are set to ∞. Afterwards at all configurations `conf` where a cop has the same position as the robber, we set `cop_depth[conf` = `robber_depth[conf]` = 0. We also push all configurarions `conf` onto the queue `cop_queue`, because in Algorithm 4.3 we only need to look at the configurations where the robber depth has changed.

Algorithm 4.1 General basic algorithm

```
1:  function General_Algorithm(Graph G)
2:      init global: robber_depth          ▷ map: configuration → integer
3:      init global: cop_depth             ▷ map: configuration → integer
4:      init global: cop_queue                    ▷ queue: configuration
5:      init global: robber_queue                 ▷ queue: configuration
6:      init global: i ← 0                                  ▷ integer
7:      Initialisation                                ▷ Algorithm 4.2
8:      while not cop_queue.empty() do
9:          i ← i +1
10:         Cop_Loop                                  ▷ Algorithm 4.3
11:         i ← i +1
12:         Robber_Loop                               ▷ Algorithm 4.4
13:     end while
14:     return Compute_Capture_Time                   ▷ Algorithm 4.5
15: end function
```

Algorithm 4.2 General basic algorithm; Initialization

```
1:  function Initialisation
2:      for all Configurations conf ∈ V^{k+1} do
3:          cop_depth[conf] ← ∞
4:          robber_depth[conf] ← ∞
5:      end for
6:      for all Configurations conf ∈ V^{k+1} where the Robber is caught do
7:          cop_depth[conf] ← 0
8:          robber_depth[conf] ← 0
9:          cop_queue.push(conf)
10:     end for
11: end function
```

Cop Loop In Algorithm 4.3 we look at all the configurations `conf` on `cop_queue`. A configuration is on `cop_queue` if in the last subroutine a new value was assigned to its robber depth. For all configurations `next_conf` that can be reached by moving a cop move from `conf`, Algorithm 4.3 checks if the cop depth is finite. If `cop_depth[next_conf]` $= \infty$, a new value gets assigned. The value that gets assigned is `robber_depth[conf]` plus one. Then `next_conf` gets pushed onto the `robber_queue`.

Algorithm 4.3 General basic algorithm; Cop loop

```
1: function Cop_Loop
2:     while not cop_queue.empty() do
3:         conf ← cop_queue.front()
4:         cop_queue.pop()
5:         depth ← robber_depth[conf] +1
6:         for all configurations next_conf that the Cops can move to from
   conf do
7:             if cop_depth[next_conf] = ∞ then
8:                 cop_depth[next_conf] ← depth
9:                 robber_queue.push(next_conf)
10:            end if
11:        end for
12:    end while
13: end function
```

Robber Loop Contrary to Algorithm 4.3 we cannot just set the `robber_depth` for every configuration reachable via a robber move from a configuration on the queue. The robber depth gets only assigned a new values if all the configuration that are reachable via a robber move have a cop depth value that is finite. Therefore Algorithm 4.4 will go through all configurations `next_conf` that the Robber can move to from `conf`. Furthermore it will look at all configurations `local_conf` that the Robber can move to from `next_conf`. Moreover Algorithm 4.4 will take these configurations `local_conf` and look at their cop depth. If for a configuration `next_conf` all values `cop_depth[local_conf]` are finite, we assign the maximum of those values plus one to `robber_depth[next_conf]`. All `next_conf` that got assigned a new robber depth get pushed onto `cop_queue`.

On the Values Stored in the Maps Every time a finite value gets stored into a depth map the value inserted is equal to `i`. In Algorithm 4.2 this is trivially true because all the values inserted are zero. Algorithm 4.3 and Algorithm 4.4 need to be looked at together, because to set their values they need values set in the other algorithm. While in Algorithm 4.3 we always insert values that are one larger than values inserted in the previous call of Algorithm 4.4. On the other

Algorithm 4.4 General basic algorithm; Robber loop

```
1: function Robber_Loop
2:     while not robber_queue.empty() do
3:         conf ← robber_queue.front()
4:         robber_queue.pop()
5:         for all configurations next_conf that the Robber can move to from
   conf do
6:             depth ← −∞
7:             for all configurations local_conf that the Robber can move to
   from next_conf do
8:                 depth ← max{depth, cop_depth[local_conf]}
9:             end for
10:            if depth < ∞ then
11:                robber_depth[next_conf] ← depth +1
12:                cop_queue.push(next_conf)
13:            end if
14:        end for
15:    end while
16: end function
```

hand Algorithm 4.4 will insert values that are one larger than one value from the last time Algorithm 4.3 was called, or the any other value inserted in a cop map plus one. Therefore in each call of either Algorithm 4.3 or Algorithm 4.4 the value inserted into the depth maps will always be `i`. This could be proven in more detail with an induction. But that proof would be needlessly complex for this little statement, so it gets omitted here.

Capture Time Algorithm 4.5 is where all the depth values get evaluated and the capture time calculated. To compute the capture time first the algorithm iterates through all the cop configurations. For each cop configuration a local maximum will be computed, it is stored in `local_capture_time`. To get this local maximum for a cop configuration we maximize the value `cop_depth[conf]`, from every configuration that is the copfiguration and any position for the robber. Then we take the minimum of all `local_capture_time` values. This minimum is the capture time and will be returned by Algorithm 4.5.

4.2 Improvements to the Algorithm

There are some things that can be done to significantly improve the runtime and space used of this algorithm, resulting in Algorithm 4.6. The first thing that can be done is that instead of inserting the values ∞ at every configuration of this

Algorithm 4.5 General basic algorithm; Capture time computation

```
1: function Compute_Capture_Time
2:     capture_time ← ∞
3:     for all Cop configurations cop_conf do
4:         init: conf = (robber, cops)                    ▷ Configuration
5:         conf.cops ← cop_conf
6:         local_capture_time ← 0
7:         for all nodes u do
8:             conf.robber ← u
9:             local_capture_time ← max{ local_capture_time,
   cop_depth[conf]}
10:        end for
11:        capture_time ← min{catching depth, local_capture_time}
12:    end for
13:    return capture_time
14: end function
```

map, we only insert values, if it is a finite one. Therefore anywhere we would check if the value of a map was infinite, we now check if the value is mapped. This helps us by saving some space. An other small improvement that can be made is that we use the value `i` declared in Algorithm 4.1. This makes the assigning of new depth values a little bit easier. This change also makes it obvious that the values inserted assigned to the cop and robber depths are only getting larger.

Because the values inserted into the cop maps are only getting larger, the capture time is possible to compute without computing the cop depth values of every single configuration. When a cop configuration is found, for that all configurations, which are achieved by choosing any robber position, have a determined cop depth. The first time this happens for a cop configuration, we already know the capture time. Therefore we use a counter map from cop configurations to integers. And each time a configuration get a cop depth value, the corresponding cop configuration gets its corresponding counter increased by one. And if one of these counters reaches the number of nodes in the graph, we know the capture time and can just return it. If this never happens, the robber is able to evade getting captured for ever.

An other modification that can be done is if we look at the cop configurations. It is straightforward to see that if in a configuration one cop is in node u and an other in node v that if the cops are switched, it has the same capture time as the other configuration. This we can use in the mapping, if we regard two configurations as the same if the cop configurations of the two only varies by a permutation. What still needs to be allowed is that multiple cops are at the same node of the graph, because there are examples of graphs where this is needed for the best cop strategy, one is given in the result section. To implement this we

don't need to change the algorithm itself, but rather the mapping, the easiest solution is to sort the cop configuration before looking up where to access the map. Sorting takes time, but because many configurations will be the same, so the overall runtime will still improve. Furthermore less space will be needed.

Algorithm 4.6 General improved algorithm

```
1:  function General_Algorithm(Graph G)
2:      init global: robber_depth   ▷ map using symmetry: configuration → int
3:      init global: cop_depth      ▷ map using symmetry: configuration → int
4:      init global: counter    ▷ map using symmetry: cop configuration → int
5:      init global: cop_queue                          ▷ queue: configuration
6:      init global: robber_queue                       ▷ queue: configuration
7:      init: depth ← 0                                                 ▷ int
8:      init: local_result                                              ▷ int
9:      Initialisation                                  ▷ Algorithm 4.7
10:     while not cop_queue.empty() do
11:         depth ← depth +1
12:         local_result ← Cop_Loop(depth)              ▷ Algorithm 4.8
13:         if local_result ≠ ∞ then
14:             return local_result
15:         end if
16:         depth ← depth +1
17:         Robber_Loop(depth)                          ▷ Algorithm 4.9
18:     end while
19:     return ∞
20: end function
```

Algorithm 4.7 General improved algorithm; Initialization

```
1:  function Initialisation
2:      for all Cop configurations cop_conf ∈ V^k do
3:          counter[cop_conf] ← 0
4:      end for
5:      depth ← 0
6:      for all Configurations conf ∈ V^{k+1} where the Robber is caught do
7:          cop_depth[conf] ← depth
8:          robber_depth[conf] ← depth
9:          cop_queue.push(conf)
10:     end for
11: end function
```

Determing the Cop Number Algorithm 4.6 can be used to not only calculate the capture time, but the cop number of a graph (Definition 2.6). This is

Algorithm 4.8 General improved algorithm; Cop loop

```
1: function Cop_Loop(depth)
2:     while not cop_queue.empty() do
3:         conf ← cop_queue.front()
4:         cop_queue.pop()
5:         for all configurations next_conf that the Cops can move to from
   conf do
6:             if next_conf ∉ cop_depth then
7:                 counter[next_conf.cops] ← counter[next_conf.cops] +1
8:                 cop_depth[next_conf] ← depth
9:                 robber_queue.push(next_conf)
10:                if counter[next_conf.cops]= V then
11:                    return depth
12:                end if
13:            end if
14:        end for
15:    end while
16: end function
```

Algorithm 4.9 General improved algorithm; Robber loop

```
1: function Robber_Loop(depth)
2:     while not robber_queue.empty() do
3:         conf ← robber_queue.front()
4:         robber_queue.pop()
5:         for all configurations next_conf that the Robber can move to from
   conf do
6:             all_in_map ← true
7:             for all configurations local_conf that the Robber can move to
   from next_conf do
8:                 if local_conf ∉ robber_depth then
9:                     all_in_map ← true
10:                    break loop
11:                end if
12:            end for
13:            if all_in_map then
14:                robber_depth[next_conf] ← depth
15:                cop_queue.push(next_conf)
16:            end if
17:        end for
18:    end while
19: end function
```

done by not just computing the capture time for a a single number of cops, but for different amounts of cops. One starts by calculating the capture time with $k \in \{1, 2, \ldots\}$ cops. The first k for which Algorithm 4.6 returns a finite value, is the cop number of the graph.

Run time We have the graph $G = (V, E)$ with k cops, where we define $n = |V|$. The graph is connected, and the maximal number of neighbours a node has is $\Delta(G)$. There are $n\binom{n}{k}$ distinct configurations for this graph and $\binom{n}{k}$ distinct cop configurations. Therefore in the initialisation, where we need to add each cop configuration with up to ke positions of the robber. Resulting in at most $2k\binom{n}{k}$ mappings that need to be done, because the values get added to both maps. Since the mapping can be done in $O(k \log k)$ time (because we sort the cop configuration), the time needed to initialize is $\left(\binom{n}{k} k^2 \log k\right)$.

Each configuration can land once in the cop queue and once in the robber queue. For an element of the cop queue we go through all the possible configurations the cops could reach, and since every cop can either stay still or move to an neighbour, this are $(\Delta(G) + 1)^k$ configurations. The mapping and the adding to the counter can be done in $O(k \log k)$ time, therefore the time it takes at most for all elements ever to be in the cop queue is $O\left(n\binom{n}{k}(\Delta(G))^k k \log k\right)$. On the other hand there is the robber queue, into which the same amount of configurations could be possibly inserted. But for a configuration inserted we need to look at all the configurations the robber could reach twice, therefore for each configuration on the robber queue the insertion into the maps is done at most $O(\Delta(G))^2$ times. Concluding that robber queue interactions take $O\left(n\binom{n}{k}(\Delta(G))^2 k \log k\right)$ time to perform. Furthermore the total time needed to run Algorithm 4.6 is $O\left(n\binom{n}{k} k \log k((\Delta(G))^k + (\Delta(G))^2)\right)$. The runtime is exponential in the number of cops, therefore limiting the usability of the algorithm for bigger numbers of cops.

Space We take the same graph we described in the run time calculation of Algorithm 4.6. To save a configuration we need all the positions of all the players this takes $O(k)$ space. In all the queues each configuration could be saved leading to $O\left(km\binom{n}{k}\right)$ space. The counter map maps every cop configuration to a value this needs $O\left(kn\binom{n}{k}\right)$ space. Finally the depth maps need $O\left(k\frac{n^{k+1}}{k!}\right)$ space. So in total this algorithm needs $O\left(k\frac{n^{k+1}}{k!}\right)$ space to save all the information.

Chapter 5

Results

5.1 Restricted Algorithm

With Algorithm 3.1 we wanted to show it is always possible for the cops to capture the robber, on the graphs constructed in Section 3.1, if the cops stick to their strategy. This is reinforced with the data we get from the computations that can be done with Algorithm 3.1. In Table 5.1 there is a listing of outputs we get from the algorithm, along with the time it took to compute them. The data from this table is also used to make two plots. In the first one the output is plotted with respect to n. Then there is made a cubic fitting of the output, which results in Figure 5.1. Theorem 3.1 claims that in the graph family with two cops, the capture time is $O(n^3)$. The results of the computation support that theorem, by delivering an upper bound that has is also growing with $O(n^3)$.

The other plot is showing the time it took to compute the output. The run time that was calculated was $O(\Delta(G)n^3)$. When we increase the component size $\hat{n}$ we do not increase the number of neighbours most nodes have, therefore the computation time should grow cubic in the number of nodes in the graph. This claim is also reinforced by the data in Figure 5.2. Furthermore we can see from this basic fitting that the constant is small.

The same thing for three cops is represented in Table 5.2, Figure 5.3 and Figure 5.4. But instead of a cubic fitting, they were fitted with a polynomial of degree four.

The last plot for Algorithm 3.1 shows the output for different amounts of cops. The output is not shown relative to the graph size, but rather to the component size $\hat{n}$. This is done to better compare the outputs of different number of cops, since the size of the graph differs strongly with different amounts of cops. And as can be seen in Figure 5.5 the time it takes for the cops to capture a robber greatly increases with the number of cops, this is only because of how the graphs in the family get adjusted to more cops. When more cops are added on the same graph it is straight forward to see that the capture time is smaller or equal to the old one. All the values used in this plot are listed in Table 5.3.

$\hat{n}$	n	capture time	comp time	$\hat{n}$	n	capture time	comp time
3	399	17	0.010947 s	78	699	210919	127.935 s
6	411	103	0.029023 s	81	711	236203	151.474 s
9	423	331	0.0567 s	84	723	263431	173.665 s
12	435	775	0.137684 s	87	735	292675	216.557 s
15	447	1507	0.266877 s	90	747	324007	258.253 s
18	459	2599	0.497294 s	93	759	357499	258.42 s
21	471	4123	0.849087 s	96	771	393223	293.061 s
24	483	6151	1.36235 s	99	783	431251	333.151 s
27	495	8755	2.12666 s	102	795	471655	378.416 s
30	507	12007	3.36106 s	105	807	514507	426.114 s
33	519	15979	4.71625 s	108	819	559879	479.676 s
36	531	20743	6.31138 s	111	831	607843	538.573 s
39	543	26371	8.33522 s	114	843	658471	601.969 s
42	555	32935	11.1916 s	117	855	711835	671.422 s
45	567	40507	14.9008 s	120	867	768007	746.586 s
48	579	49159	18.5962 s	123	879	827059	866.226 s
51	591	58963	23.3386 s	126	891	889063	992.073 s
54	603	69991	29.1229 s	129	903	954091	1161.64 s
57	615	82315	36.0223 s	132	915	1022215	1289.87 s
60	627	96007	44.0484 s	135	927	1093507	1368.15 s
63	639	111139	53.5215 s	138	939	1168039	1408.84 s
66	651	127783	64.8244 s	141	951	1245883	1491.65 s
69	663	146011	76.3598 s	144	963	1327111	1602.22 s
72	675	165895	91.686 s	147	975	1411795	1748.22 s
75	687	187507	108.184 s	150	987	1500007	1936.39 s

Table 5.1: Results of the algorithm, run on graphs from the family with 2 cops

$\hat{n}$	n	capture time	comp time	$\hat{n}$	n	capture time	comp time
3	459	17	0.013704 s	33	609	175699	67.8899 s
6	474	199	0.053816 s	36	624	248839	103.322 s
9	489	979	0.226422 s	39	639	342739	151.036 s
12	504	3079	0.712995 s	42	654	460999	215.364 s
15	519	7507	2.01237 s	45	669	607507	299.362 s
18	534	15559	4.40898 s	48	684	786439	405.962 s
21	549	28819	8.87236 s	51	699	1002259	547.792 s
24	564	49159	16.1577 s	54	714	1259719	726.715 s
27	579	78739	27.7676 s	57	729	1563859	936.544 s
30	594	120007	44.4556 s	60	744	1920007	1218.48 s

Table 5.2: Results of the algorithm, run on graphs from the family with 3 cops

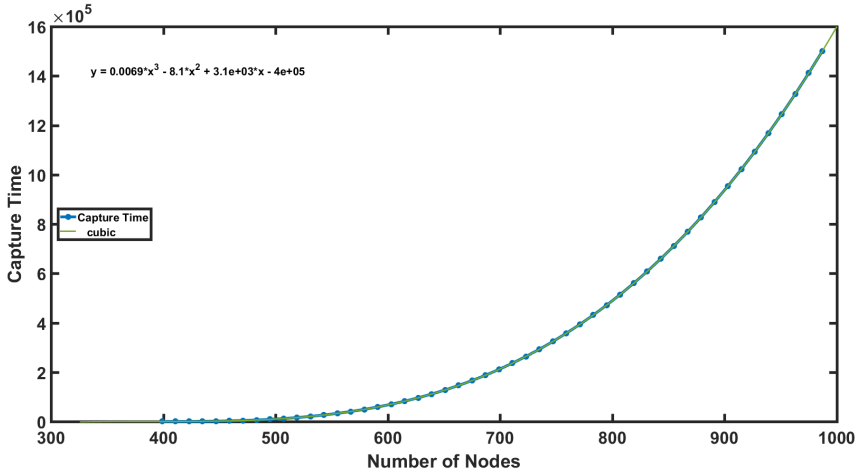


Figure 5.1: Computed capture time with cubic fitting, on graphs with 2 cops; Exact values found in Table 5.1

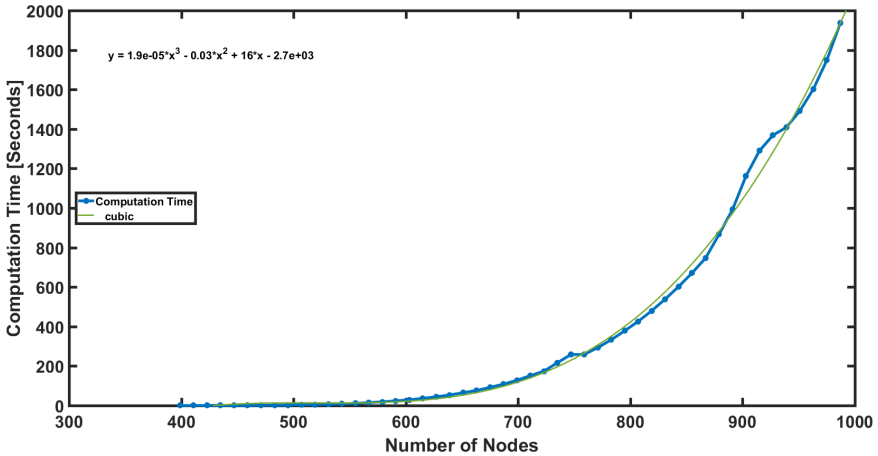


Figure 5.2: Computation time with cubic fitting, on graphs with 2 cops; Exact values found in Table 5.1

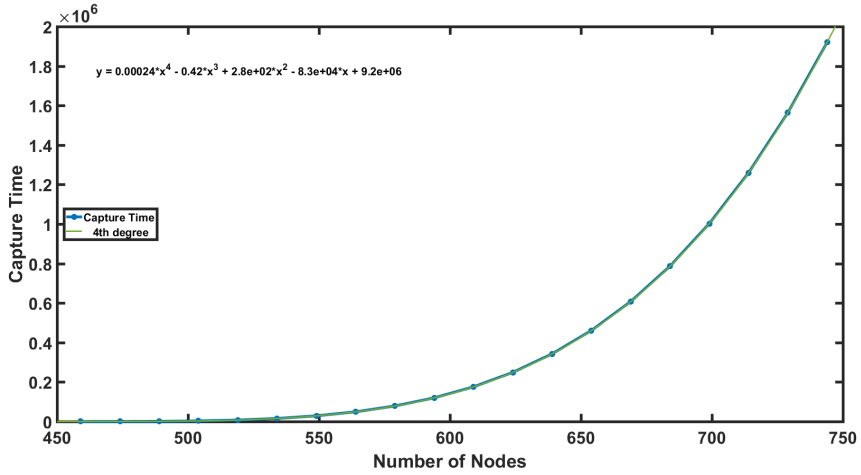


Figure 5.3: Computed capture time with polynomial fitting, on graphs with 3 cops;
Exact values found in Table 5.2

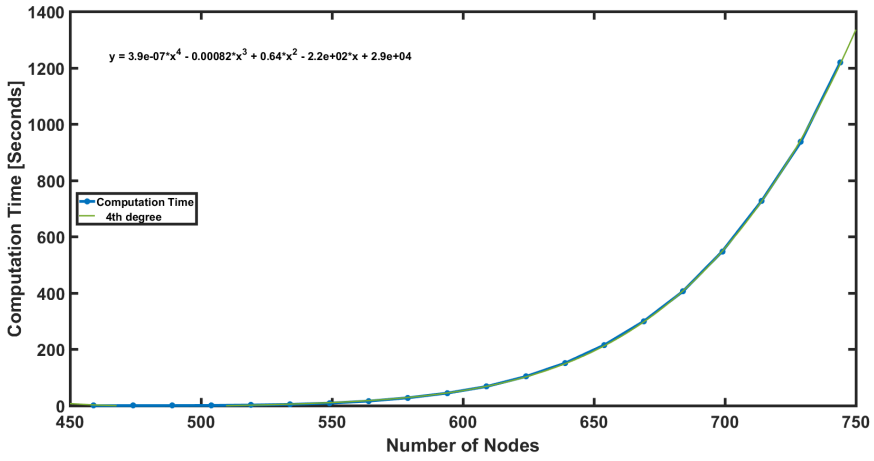


Figure 5.4: Computation time with polynomial fitting, on graphs with 3 cops;
Exact values found in Table 5.2

$\hat{n}$	k	n	output	comp time	$\hat{n}$	k	n	output	comp time
3	2	399	17	0.006772 s	3	4	607	17	0.016935 s
6	2	411	103	0.015173 s	6	4	625	391	0.110034 s
9	2	423	331	0.040383 s	9	4	643	2923	0.805018 s
12	2	435	775	0.101146 s	12	4	661	12295	4.06905 s
15	2	447	1507	0.235818 s	15	4	679	37507	11.178 s
18	2	459	2599	0.401265 s	18	4	697	93319	30.1597 s
21	2	471	4123	0.669972 s	21	4	715	201691	70.0834 s
24	2	483	6151	1.14168 s	24	4	733	393223	145.858 s
27	2	495	8755	1.70783 s	27	4	751	708595	287.077 s
30	2	507	12007	2.42001 s	30	4	769	1200007	523.385 s
3	3	459	17	0.009987 s	3	5	879	17	0.023656 s
6	3	474	199	0.048311 s	6	5	900	775	0.259447 s
9	3	489	979	0.183818 s	9	5	921	8755	2.83049 s
12	3	504	3079	0.604732 s	12	5	942	49159	17.695 s
15	3	519	7507	1.47623 s	15	5	963	187507	74.6015 s
18	3	534	15559	3.33862 s	18	5	984	559879	241.917 s
21	3	549	28819	6.71791 s	21	5	1005	1411795	651.378 s
24	3	564	49159	15.1171 s	24	5	1026	3145735	1734.89 s
27	3	579	78739	22.9673 s	27	5	1047	6377299	3392.86 s
30	3	594	120007	37.071 s	30	5	1068	12000007	6397.39 s

Table 5.3: Results of the algorithm, run on graphs from the family with different amount of cops; The output is the capture time

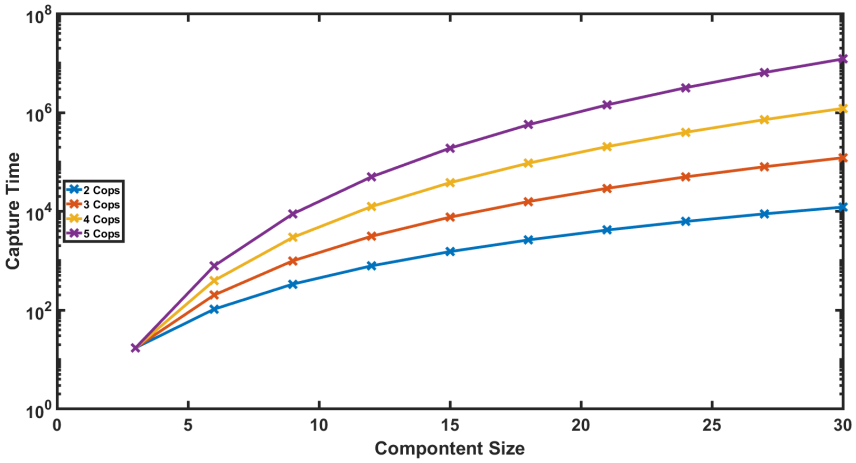


Figure 5.5: Computed capture time on graphs with different cops; The component size is $\hat{n}$; The capture time is scaled logarithmically Exact values found in Table 5.3

$\hat{n}$	n	capture time	computation time
3	399	11	727.686 s
6	411	51	944.851 s
9	423	159	1316.14 s
12	435	351	1866.51 s
15	447	673	2699.42 s
18	459	1161	3582.87 s
21	471	1851	4899.48 s
24	483	2779	6774.52 s
27	495	3981	9716.61 s

Table 5.4: Results of the algorithm, run on graphs from the family with 2 cops

When we analyse the numbers of the outputs, we get that the following:

$$x = 12\left(\frac{\hat{n}}{3}\right)^{k+1} + 7 \text{ for } \hat{n} > 3$$

where x is the output of our algorithm. But for $\hat{n} = 3$ we always get that $x = 17$. This is nice, since this corresponds to the numbers that were assumed.

5.2 General Algorithm

Algorithm 4.6 computes the capture time of any graph. But since the algorithm performs much slower, we can not get results from graphs of the same size. The performance gets so bad that for the smallest instance with three cops, the computation would take approximately a month. The computation would also need approximately 128 Gb of RAM. Because of these reasons it does not make much sense to try an instance with three cops. With only two cops there at least some instances that can be calculated exactly. The results of those calculations are listed in Table 5.4. Furthermore this is illustrated in Figure 5.6 and Figure 5.7.

And since the values are all roughly half what the values for a fixed cop strategy are, we assume that at least for two cops this would continue for larger graphs. With this information not only the upper bound is $O(n^3)$ but also the actual capture time is in $O(n^3)$. And thus we calculated some more evidence of the correctness of Theorem 3.1.

A small example of such an optimal path is illustrated in Figure 5.8. In those pictures the edges of the graph are left out for simplicity. Instead of drawing all the edges, all the neighbours of cops are circled red, and the neighbours of robbers are lightly blue filled. The nodes, the cops are on, are coloured red and those the robbers are on blue. For each configuration on the path to capturing

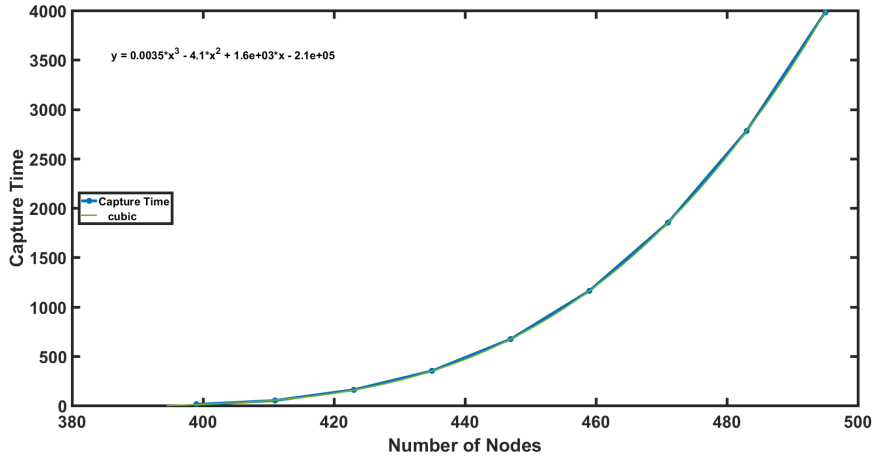


Figure 5.6: Computed capture time with cubic fitting, on graphs with 2 cops; Computed with the general algorithm; Exact values found in Table 5.4

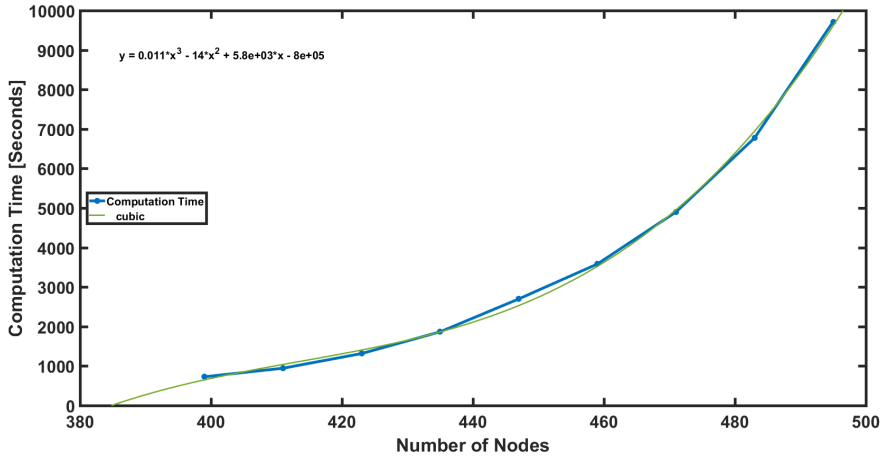


Figure 5.7: Computation time with cubic fitting, on graphs with 2 cops; Computed with the general algorithm; Exact values found in Table 5.4

the robber, one figure is made showing the positions of each figure and the nodes they could move to.

5.3 Examples of Cop Numbers

The general algorithm can also be used to compute the cop number of any graph. The first example is the cyclic graph, which has cop number two if it has at least four nodes. For 30 nodes the cyclic graph is shown in Figure 5.9.

The Petersen graph [3] is the smallest graph with cop number three. This is also confirmed by the computation with the algorithm. The Petersen graph is shown in Figure 5.10.

In Figure 5.11 an example of a graph, which requires multiple cops to have the same position to capture the robber as fast as possible, is shown. This graph has cop number two.

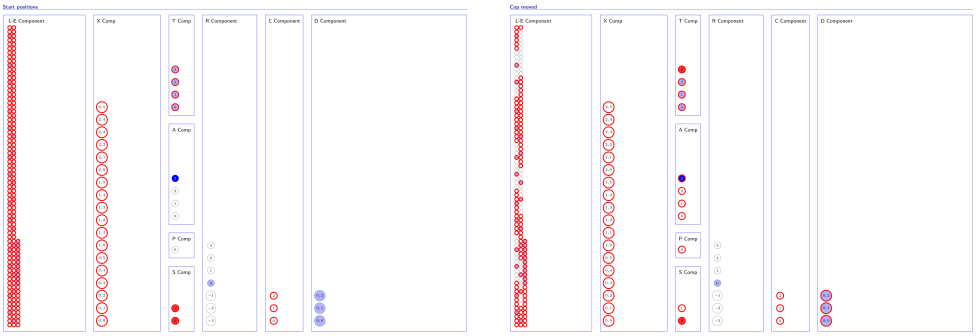
The Robertson-Wenger graph [7] is shown in Figure 5.12. With Algorithm 4.6 it is possible to compute that the cop number of the Robertson-Wenger graph is five. But computing this result took approximately 14 hours. Because the Robertson-Wenger graph is one of the smallest graphs with cop number five, it takes too long to compute the capture time for most other graphs with cop number five. Furthermore more it would even take considerably longer for a graph with cop number six, since the computation time is exponential in the number of cops.

The last graph that is presented here is a two dimensional torus. This graph requires three cops to capture the robber. The torus is drawn in Figure 5.13.

5.4 How Many Cops Are Needed

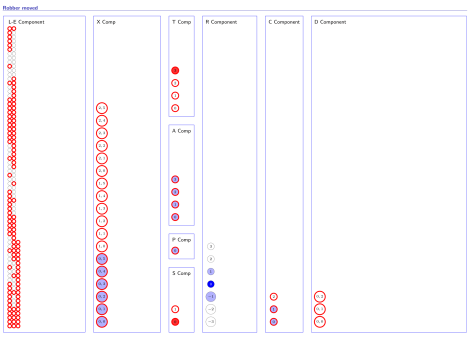
There is an unproven conjunction of Meyniel [4] that the cop number of a graph is in $O(\sqrt{n})$. With the help of Theorem 2.7 we can construct such graphs that need roughly $\sqrt{n}$ cops to capture the robber. Such a construction might be useful to construct other graph families, similar to the one in Section 3.1 the component $G(\mathcal{E}, \mathcal{L})$. But there an other construction was used. The construction used in $G(\mathcal{E}, \mathcal{L})$ is based on projective planes [2]. So why would we need another construction? The answer to that lies in the number of nodes used to create a graph. Our new approach need roughly k^2 nodes to create a graph with cop number k. On the other hand projective planes need around $2k^2$. Also this new construction might give us insights on the conjunction of Meyniel.

The construction describes a graph that k cops are needed to capture the robber, the number of nodes need is $k^2 + k$. Theorem 2.7 states that we have a graph, where there are no cycles of length smaller than five and if all the nodes

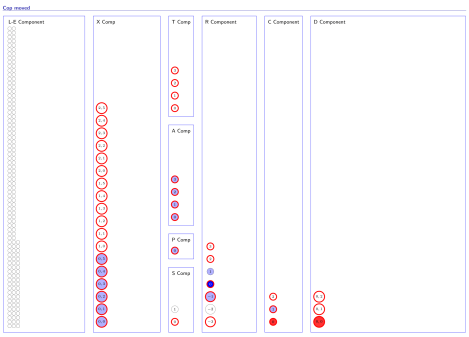


(a) Initialisation

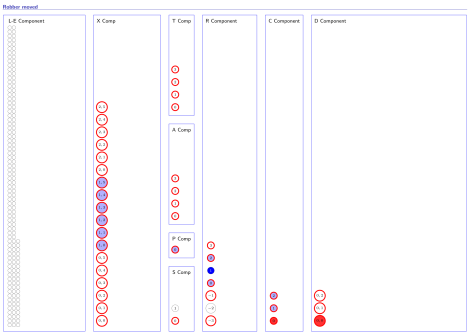
(b) First cop move



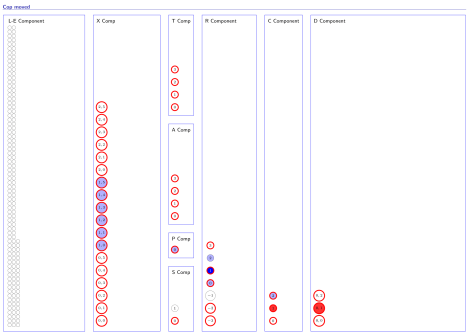
(c) First robber move



(d) Second cop move

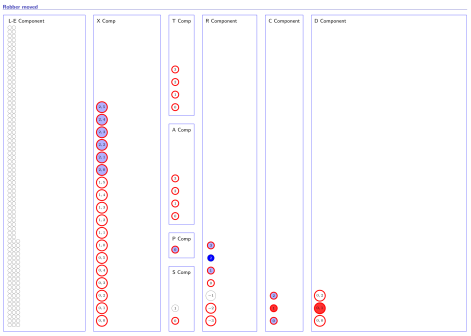


(e) Second robber move

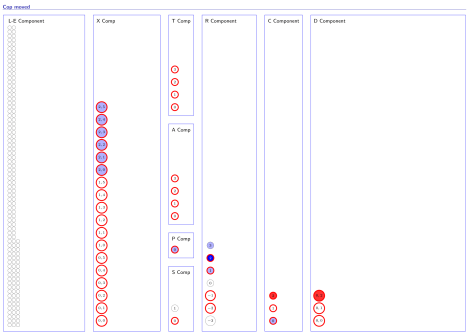


(f) Third cop move

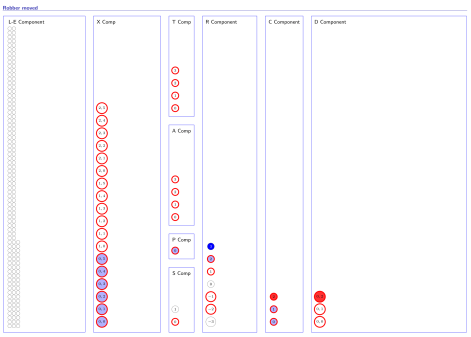
Figure 5.8: An Optimal Path, Part 1



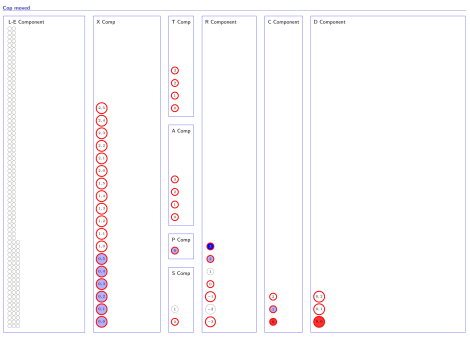
(g) Third robber move



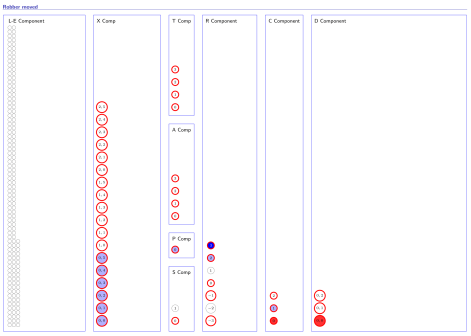
(h) Fourth cop move



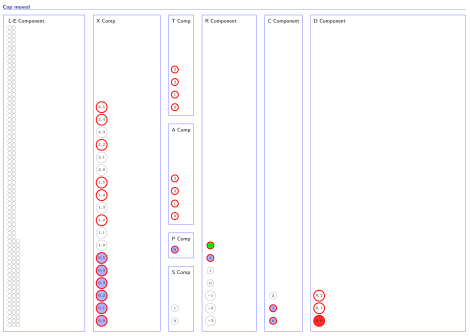
(i) Fourth robber move



(j) Fifth cop move



(k) Fifth robber move



(l) Cops capture the robber

Figure 5.8: An Optimal Path, Part 2

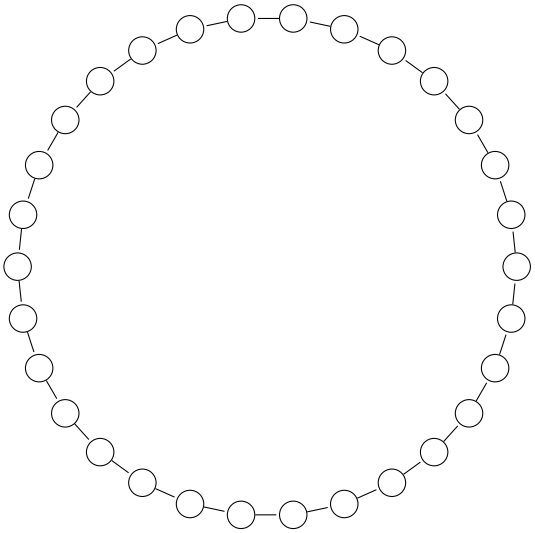


Figure 5.9: Cyclic graph C_{30} with cop number 2 and capture time 13

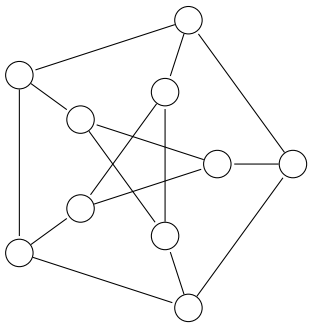


Figure 5.10: Petersen graph with cop number 3 and capture time 1

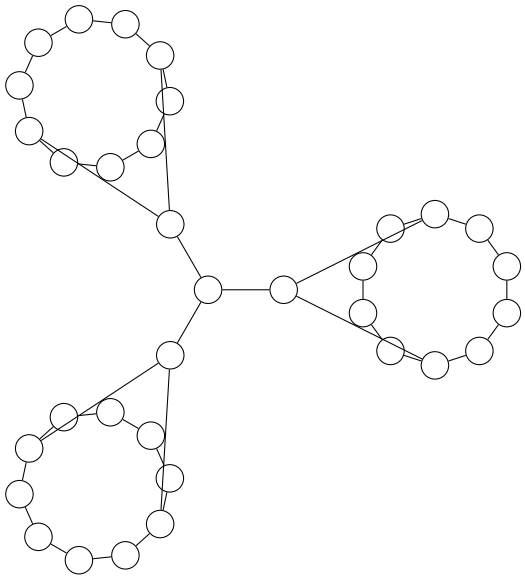


Figure 5.11: Graph where both optimal playing cops start in the middle node, it has cop number 2 and capture time 7

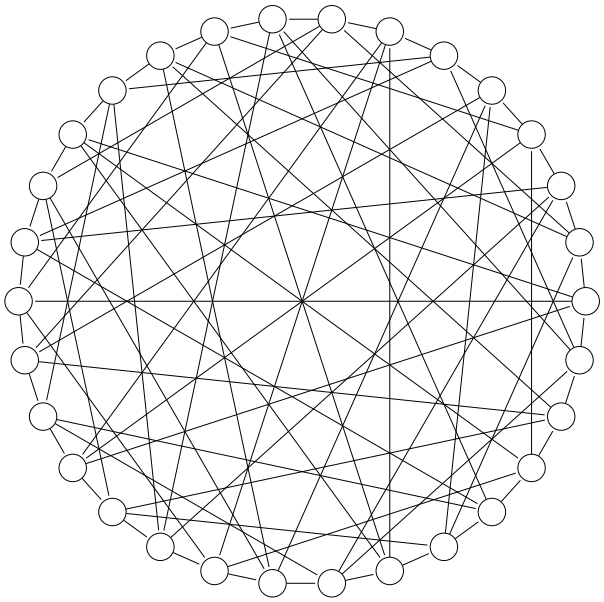


Figure 5.12: Robertson-Wenger Graph with cop number 5 and capture time 3

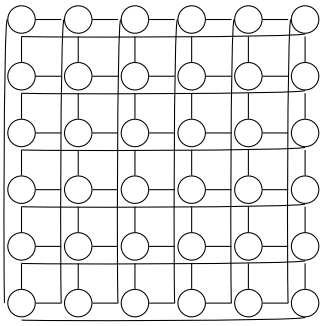


Figure 5.13: Torus with 36 nodes, cop number 3 and capture time 7

k	Number of Possibilities
2	1
3	2
4	24
5	9'953'280

Table 5.5: Number of Ways to Connect $\mathcal{V}$

have at least k neighbours, at least k are needed to capture the robber on the graph. The construction, it is represented in Figure 5.14 , starts with the node O. It is connected to k nodes. The nodes O is connected to, are called $L_0, \ldots, L_k$. For each i from 0 to k L_i is connected to $v_i^0, \ldots, v_i^{k-1}$. Furthermore the node R^j, for j from 0 to $k-1$, is connected to the nodes $v_0^j, \ldots, v_k^j$. The set $\mathcal{V}$ is the set of all the nodes v_i^j, for $i \in \{0, \ldots, k\}$ and $j \in \{0, \ldots, k-1\}$. The nodes O, L_i and R^j, for $i \in \{0, \ldots, k\}$ and $j \in \{0, \ldots, k-1\}$, have k neighbours. But the nodes v_i^j not yet. But to use Theorem 2.7 it is required for them to also have k neighbours.

The rest of those edges could for example be chosen by brute force. Each of the nodes in $\mathcal{V}$ need $k-2$ more nodes. When choosing these edges one has to follow two rules. The first rule states that in the component $\mathcal{V}$ there is no cycle of length smaller than five. The second rule states that when v_i^j is connected to $v_{i_1}^{j_1}$ and $v_{i_2}^{j_2}$ with $v_{i_1}^{j_1} \neq v_{i_2}^{j_2}$ then neither $i_1 = i_2$ nor $j_1 = j_2$ is allowed. The second rule assures that no cycle too short is made either over a node L_i or R^j. With this the number of ways these edges could be arranged is shown in Table 5.5. Since these numbers are growing fast, it is reasonable to assume that this construction can be done for any number of cops. It even might be possible to figure out an explicit way to choose those edges, making the construction much easier.

Figure 5.15 shows a graph constructed in that way, but it was rearranged onto a cycle. It was rearranged to improve the visibility of the individual edges. To create to create a graph with $k = 4$ a possibility of arranging the additional

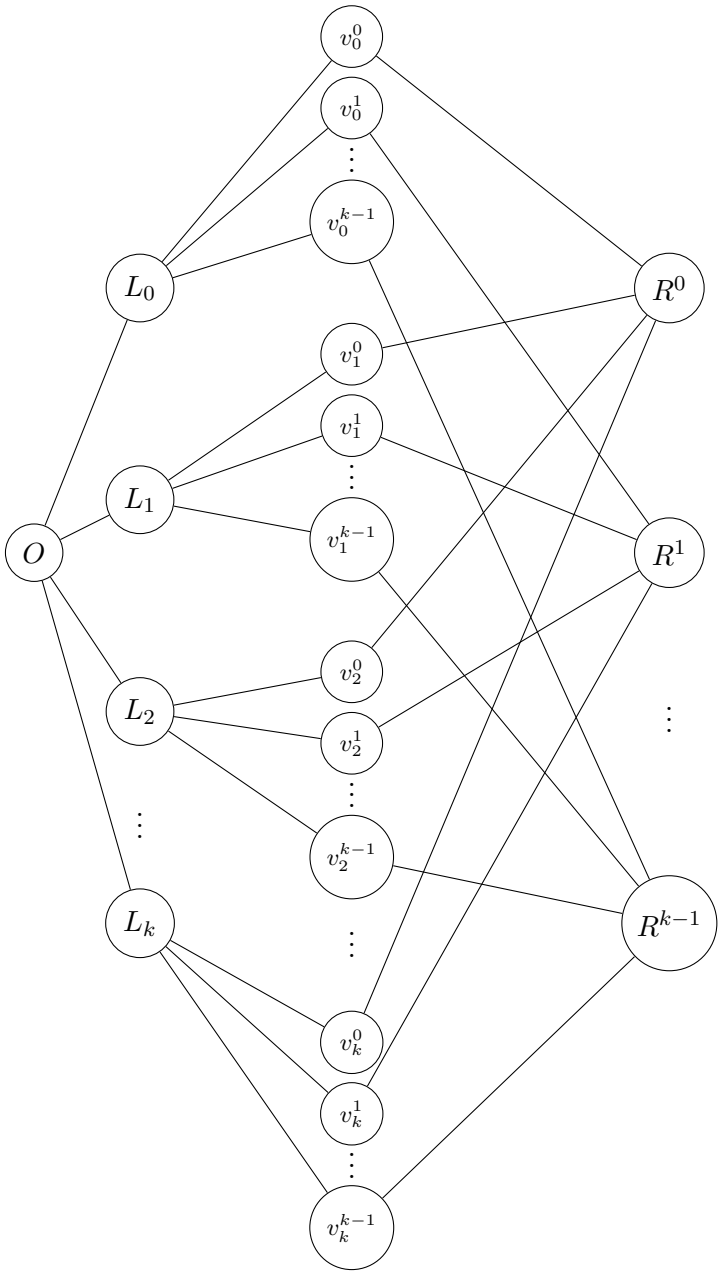


Figure 5.14: Construction of a graph with high cop number

node	edges to
i_0^0	i_1^1 and i_2^2
i_0^1	i_1^2 and i_2^0
i_0^2	i_1^0 and i_2^1
i_1^0	i_0^2 and i_3^1
i_1^1	i_0^0 and i_3^2
i_1^2	i_0^1 and i_3^0
i_2^0	i_0^1 and i_3^2
i_2^1	i_0^2 and i_3^0
i_2^2	i_0^0 and i_3^1
i_3^0	i_1^2 and i_2^1
i_3^1	i_1^0 and i_2^2
i_3^2	i_1^1 and i_2^0

Table 5.6: List of the brute forced edges, for $k = 4$

edges is shown in Table 5.6.

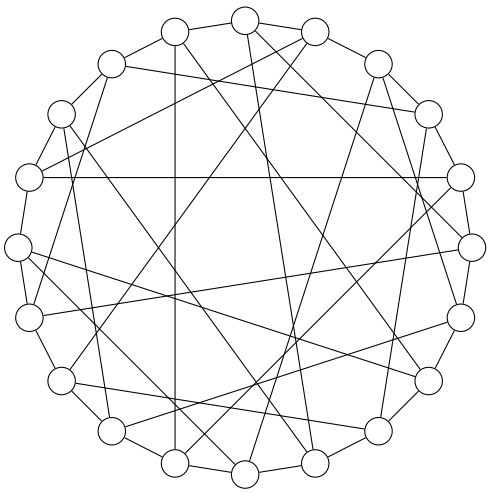


Figure 5.15: Graph with Cop Number 4

Bibliography

[1] M. Aigner and M. Fromme. A game of cops and robbers. *Discrete Applied Mathematics*, Volume 8, 1984. https://doi.org/10.1016/0166-218X(84)90073-8.

[2] Anthony Bonato. What is cop number, 2012. http://www.ams.org/notices/201208/rtx120801100p.pdf.

[3] Andries E. Brouwer. The petersen graph. http://www.win.tue.nl/~aeb/drg/graphs/Petersen.html.

[4] Meyniel. Meyniels conjuncture. https://arxiv.org/abs/1308.3385.

[5] S. Brandt, Y. Emek, J. Uitto and R. Wattenhofer. A tight lower bound for the capture time of the cops and robbers game, 2017. https://ie.technion.ac.il/~yemek/Publications/tlbctcrg.pdf.

[6] George T. Heineman; Gary Pollice; Stanley Selkow. *The Elements of Typographic Style*. Oreilly Media, 2008. ISBN 978-0-596-51624-6.

[7] Eric W. Weisstein. Robertson-wegner graph. From MathWorld–A Wolfram Web Resource.http://mathworld.wolfram.com/Robertson-WegnerGraph.html.