University of Alberta

GAME TREE SEARCH ALGORITHMS FOR THE GAME OF COPS AND ROBBER

by

Carsten Moldenhauer

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Fall 2009
Edmonton, Alberta

Examining Committee

Jonathan Schaeffer, Computing Science

Nathan Sturtevant, Computing Science

Martin Müller, Computing Science

Mike Kouritzin, Mathematical and Statistical Sciences

Abstract

Moving target search has been given much attention during the last twenty years. It is a game in which multiple pursuers (cops) try to catch an evading agent (robber) and also known as the game of cops and robber. Within this thesis we study a discrete alternating version played on a graph with given initial positions for the cops and the robber, providing a number of results for optimal and sub-optimal approaches to the game.

We review the mathematical definition and investigate an optimal algorithm to solve the entire state space. This retrograde analysis algorithm can also be used to compute a best response against a given robber algorithm.

Furthermore, we study algorithms to compute solutions to the game for given initial positions. We present several methods: Two-Agent IDA*, Proof-Number Search, Alpha-Beta Minimax and Reverse Minimax A*, a new algorithm. We prove their correctness when our enhancements are used and conduct experiments on multiple scenarios which establishes the first benchmarks for optimal moving target search.

Additionally, we study sub-optimal algorithms for the robber, including Cover, Dynamic Abstract Minimax, Minimax, hill climbing with distance heuristic, a random beacon algorithm, and propose two new methods named TrailMax and Dynamic Abstract TrailMax. Experiments show that our methods outperform all the other algorithms in solution quality while meeting computer game computation time constraints.

Finally, we discuss the mathematical background of the game including complexity and variations of the game. We further develop a method to solve the simultaneous action game and prove its correctness.

Table of Contents

List of Tables

List of Figures

List of Notations

$a \leftarrow b$	assignment operator, a takes the value of b
$c(G)$	cop number of graph G
$c(\mathbf{s}, \mathbf{s}')$	cost of an action that transfers state $\mathbf{s}$ into $\mathbf{s}'$
ct(G)	capture time of graph G
ct$(\mathbf{s})$	capture time of state $\mathbf{s}$
$E(G)$	set of edges of the graph G
G	graph that the game is played on
IPN	Iterative Proof-Number Search
k	number of cops present in the game
M	move graph
MTS	moving target search
n	number of vertices of the graph that the game is played on
$N(\mathbf{s})$	neighborhood of state $\mathbf{s}$ in the move graph
$N(v)$	neighborhood of vertex v in the graph
P_n	path of length n
RA	retrograde analysis
RMA*	Reverse Minimax A*
st(G)	search time of graph G
TIDA*	Two-Agent IDA*
$V(G)$	set of vertices of the graph G

Chapter 1

Introduction

Pursuit games have fascinated mankind for centuries and have been studied in many domains such as warfare, law enforcement and computer games. As the name suggests, the goal of these games is for a number of pursuers to catch a target. The pursuers, also known as *cops*, are forced to cooperate to achieve their task. The target, also called the *robber*, tries to evade capture and to complicate the pursuers' task. The game is played on domains such as maps or graphs. As an example, consider the video games Grand Theft Auto and Need For Speed. Here, the player controls a character that is pursued by the cops. These agents are computer generated, i.e. are controlled by algorithms that implement pursuit strategies. The game can also be played with interchanged roles. Consider flight simulations where the player tries to shoot or catch some targets that are computer generated. All these games have in common that the generated agents have to exhibit intelligent behavior to make the game interesting and challenging. Hence the need for strong pursuit and evasion strategies.

With the development of game theory in the 1950's the first serious scientific study of pursuit games began. The first version of the game was played in a continuous space. Here, the motion of all agents is modeled by their underlying physical processes. Continuous actions like accelerating and turning with or without constraints are described by differential equations and hence the problem becomes a game on differential equations. A classic reference for the continuous case is Isaacs' book on differential games [35].

Modeling time and space is of major importance for the definition of the game. Both can be interpreted in a continuous or discrete way, thus resulting in many different possible variations. Therefore, it is not surprising that the literature on pursuit games is very broad and cannot be summarized succinctly. The interested reader is referred to Isaacs' [35] and Nahin's [61] books as well as surveys on graph searching by Alspach [4] and Fomin [23] for an introduction and first overview.

This thesis is inspired by modern computer games. Here, time and space are usually discrete entities so as to enable feasible computation and simulation. Maps are modeled by graphs and the cops and the robber take turns in their movement. The mathematical study of this discrete problem originated in the 1980's with the introduction of the "cops and robber" game by Quillot [68, 69] and Nowakowski and Winkler [64]. Typical questions that have been addressed are the number of cops

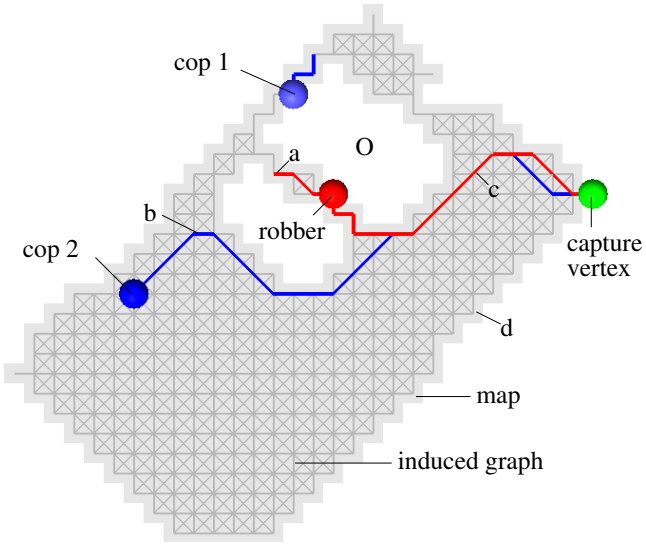


Figure 1.1: Cops and robber scenario. Two cops try to capture one robber on a given map.

required to guarantee capture of an evading robber, the time needed to capture and the complexity of computing solutions to the problem. In contrast to the mathematical study, the game first appeared in the artificial intelligence literature in the early 1990's under the name "moving target search" (MTS) as introduced by Ishida and Korf [38]. Following this initial study, the focus of MTS research has been in developing strong pursuit strategies [29, 37, 39, 44, 46, 55, 56].

As an example scenario consider Figure 1.1. The game takes place on a map that is discretized into a graph that represents the structure of the map. Here, two cops and one robber are present and all agents move at the same speed. The game is played in alternating turns with the cops beginning. The goal of the cops is to capture the robber as quickly as possible. In contrast, the robber's goal is to maximize the time to capture or, if possible, to evade the cops forever.

Due to the obstacles, i.e. unpassable white regions in the upper part of Figure 1.1, it is clear that one cop will not be sufficient for capture in this scenario because the robber could run around these obstacles. There are two players: one controlling the cops and one the robber. Assuming rationality of both players and that they will play according to the true minimax value of the game the robber will first move diagonally up left to a and force the second cop to move diagonally up right to b. Once the second cop has arrived in b the robber turns around and moves to his initial vertex. He then runs towards the indicated vertex where capture occurs being followed by the second cop. Since we assume rationality of the players the first cop does not have to move much. He only moves when the robber approaches c to equilibrate the distance between the two cops and the robber around the obstacle O.

It can be seen from the example in Figure 1.1 that good strategies for pursuit and evasion are not trivial to find. For example, if both cops pursue the robber by following a shortest path towards him, they would run on paths through a. Hence, the robber will be able to escape to c and can then

loop around obstacle O. Both cops will follow the robber's path moving along the same side of O. Thus, the robber will be able to evade the cops forever. This shows the need of coordination for the movement of the cops which is a difficult problem especially in large and complex maps.

Furthermore, if the robber runs a greedy strategy of maximizing the distance to the cops he will immediately run to d and stay there until both cops have come around the obstacles and captured him. This is highly suboptimal since the robber could achieve a longer survival time with better play. However, in contrast to the cops that want to capture the robber, there is no obvious goal location that the robber could run to. Therefore, choosing between possible run-away directions is difficult and must also take into account the cops' movements. Hence, determining good strategies for the robber is a challenging task.

In the first part of this thesis, we will develop algorithms that solve the game optimally and investigate their performance. Here, optimality means the true minimax value of the game where the cops want to minimize and the robber wants to maximize the time to capture. Despite recent research on MTS, there has been no study of algorithms for optimal MTS. In fact, known approaches are learning and anytime algorithms that try to approximate the optimal solution primarily for the cop's side. This work establishes the first systematic study of optimal algorithms and therefore yields a benchmark for the optimal cops and robber problem.

Retrograde analysis can be used to solve the entire state space, i.e. solve the discrete game as described above for every possible initial position of cops and robber, and will be discussed in Chapter 3. We further study its theoretical and practical performance and outline how this algorithm can be used to compute a best response against a given target algorithm.

The question of how optimal solutions for one initial position can be computed quickly will be addressed in Chapter 4. In this case, it is possible to use estimates on the capture time, i.e. heuristics, to speed up the computation. It will be shown that existing algorithms can be enhanced to be more efficient. Furthermore, a new algorithm, Reverse Minimax A*, will be developed that outperforms all the other approaches. Benchmarks will be given on several experiment configurations.

In today's computer games, the player often controls a robber being chased by computer generated police agents. The same game turned around, i.e. the player controlling a cop and having to chase down a computer generated robber, is the motivation for the second part of this work. The goal is to develop algorithms that compute move policies for the robber that are usable for commercial computer games. This involves solving several problems. First, we want to generate policies that are challenging for the cops. Second, computer games impose tight resource allocation constraints, i.e. computation time and memory usage. Therefore, it is not feasible to compute optimal solutions due to high computation time requirements of optimal algorithms. The intention is to generate move policies for the robber that are close to an optimal policy and are computable quickly. In Chapter 5, existing algorithms will be reviewed and two newly developed algorithms, TrailMax and Dynamic Abstract Trailmax, will be discussed. Experiments show that the new methods outperform

all the other algorithms by achieving up to 98% optimality, while meeting modern computer game computation time constraints.

Additional mathematical background and a few new results on the capture time and the characterization of k-cop-win graphs will be discussed in Chapter 6. We will further outline known results on the number of cops required to catch a robber in a graph, the search time of a graph and the complexity of computing solutions for the cops and robber problem. Moreover, we will list some common variations of the game and therefore bridge the gap between the original definition and more general pursuit evasion games.

We will study a variation of the game, the simultaneous action game in Chapter 7. Since an optimal solution will require mixed strategies, this game cannot be approached with the usual game tree search algorithms and more general methods are needed. We show how an undiscounted Markov Game formulation can be used to solve the game and prove that, under the assumption that the cops can win the game, the value iteration method converges to the optimal values.

Some of the work in this thesis has been published at the Eighth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009) [58] as well as the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI 2009) [57].

Chapter 2

Mathematical definitions

The sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work.

John Von Neumann (1903 - 1957)

Within this chapter we will define the basic mathematical terms for the game of cops and robber that will be needed in subsequent chapters. We begin with the original definition of the game that was first independently introduced by Quillot [68, 69] and Nowakowski and Winkler [64]. Afterwards, we will define the state space and move graph of the game and give a standard proof that either of the two players has a winning strategy. This proof is the motivation for the retrograde analysis algorithm in Chapter 3. We will define the cop number, search time and capture time of a graph. These terms are frequently used within the next chapters. However, an in-depth mathematical discussion of these graph properties will be delayed until Chapter 6. Furthermore, we will outline the characterization of 1-cop-win graphs here and define octile maps. The characterization is needed, amongst others, for the proof of correctness of our method TrailMax in Section 5.4. Octile maps will be used within all the experiments of the following chapters. Ideas to generalize the characterization to k-cop-win graphs are discussed in Chapter 6. Moreover, we will consider the complexity of computing solutions and variations of the original problem in Chapter 6.

For notation and terminology that is not defined here, the reader is referred to Diestel [20]. Vectors or tuples are set in bold throughout the entire thesis.

Definition 2.1 (cops and robber game)
Let G be an undirected, finite, simple graph and let k be the number of cops. Further, let C denote the player controlling the cops and R denote the player controlling the robber. The game begins with C choosing an initial vertex for each cop followed by R choosing the initial vertex for the robber. The game is played alternatingly with C beginning. A move consists of choosing new locations for each of the agents under a player's control. An agent can either move to a neighboring vertex or pass

and remain on its current one. It is allowed that multiple cops occupy the same vertex. The game is played with perfect information, hence all the agent's positions are known to both players at all times.

C's goal is to eventually move a cop onto the vertex currently occupied by the robber. If this happens the game ends and C wins. R's objective is to keep the cop from winning. If he can avoid capture forever, which means the game is of infinite duration, then R wins.

G is said to be *k-cop-win* if k cops have a winning strategy on G. A 1-cop-win graph is also referred to as *cop-win*.

Remark. Instead of letting an agent pass a turn, some publications prefer to use reflexive graphs.

Since the game of cops and robber is a two player game, player C controlling all the cops and player R controlling the robber, we will identify C with the cops and R with the robber in the following. Furthermore, we will refer to G as the graph on which the game is being played on and to k as the number of cops. We can safely assume that G is connected, otherwise all the following discussions can be extended to disconnected graphs by applying them to the component of G containing the robber.

Either of the two players has a winning strategy. By introducing some notation that will be needed in subsequent chapters we will give a proof of this statement in the following. Note that this is a common proof within the literature. We will first define the state space and the move graph of the game and then define a hierarchy that can be used to determine whether the cops or the robber have a winning strategy.

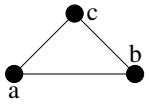
Definition 2.2 (state space and move graph)
Define S as the *state space* of the game which is the set of all $(k+2)$-tuples of possible positions of the robber and the k cops with an additional tag denoting the player to move next.

$$S = V(G) \times V(G)^k \times \{0, 1\}.$$

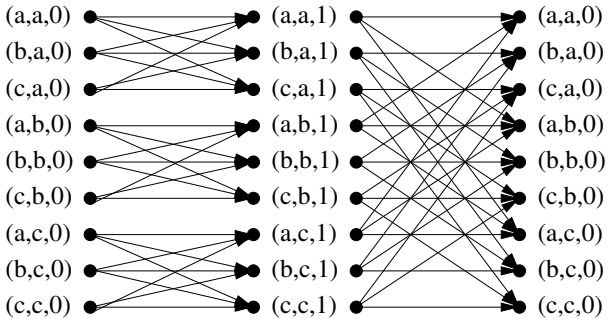
We define $\text{turn}(\mathbf{s}) = s_{k+2}$ for every state $\mathbf{s} \in S$ to specify which of the players is to move next in $\mathbf{s}$. It is the robber's turn if $\text{turn}(\mathbf{s}) = 0$ and the cop's turn otherwise. We partition the state space S into two disjoint sets. S_C denotes the set of states where the cop is to move next and S_R denotes the set of states where the robber is to move next. Hence, $S_C = \{\mathbf{s} \in S | \text{turn}(\mathbf{s}) = 1\}$ and $S_R = \{\mathbf{s} \in S | \text{turn}(\mathbf{s}) = 0\}$.

Furthermore, define $M = (S, E_M)$ as the *move graph*. The move graph has a directed edge for every possible joint move of the cops from one joint position $\mathbf{c}$ to another $\mathbf{c}'$ if it is the cops' turn, and an edge for every move the robber could make from a vertex r to r' when it is his turn. Recall that an agent is also allowed to remain on his current position. Hence

$$\begin{aligned} E_M \quad = \quad & \{((r, \mathbf{c}, 0), (r', \mathbf{c}, 1)) \mid (r, r') \in E(G) \ \vee \ r = r'\} \cup \\ & \{((r, \mathbf{c}, 1), (r, \mathbf{c}', 0)) \mid \forall i, 1 \leq i \leq k : \ (c_i, c_i') \in E(G) \ \vee \ c_i = c_i'\}. \end{aligned}$$



(a) original graph G



(b) move graph M

Figure 2.1: The move graph of a triangle. Note that each state has been copied to enable better visualization.

We depict an example of the above definitions in Figure 2.1. The original graph G, which the game is being played on, is a triangle (Figure 2.1(a)) and we play with one cop. The move graph includes a vertex for every pair of vertices of G with additional tags 0 and 1 to indicate that it is the robber's or the cop's turn. Note that all states have been copied once to enable a better visualization in Figure 2.1(b). There is a directed edge between two states $\mathbf{s}_1$ and $\mathbf{s}_2$ in the move graph if the agent, who is to play next, can transform $\mathbf{s}_1$ into $\mathbf{s}_2$ by moving on G. The move graph is bipartite due to the fact that the turn tag changes every move.

The proof that either of the two players has a winning strategy is important to this thesis because the construction that will be used will also be exploited in later chapters to develop algorithms to solve the game computationally. We will define a hierarchy of sets W_i and W_i' of states. The intention is to construct these sets such that W_i consists of all states in which the cops are to move and where the cops can win in at most i (joint) moves. Furthermore, W_i' will include all states in which the robber is to move and in which the cops can win after at most $(i-1)$ moves of the cops. Therefore, let W_0 and W_0' denote all the *terminal states* of the game, i.e. all the states where the robber is caught. Note that the last move is always by the cop since the robber can pass his turn and wait for the cop to capture him. We denote the position of the robber by r and the positions of the cops by $\mathbf{c}$.

$$\begin{aligned} W_0 &= \{(r, \mathbf{c}, 1) \mid \exists 1 \leq i \leq k : \ c_i = r\} \\ W_0' &= \{(r, \mathbf{c}, 0) \mid \exists 1 \leq i \leq k : \ c_i = r\}. \end{aligned}$$

We define the hierarchy inductively. Let W_{i+1}' be the states where the robber is to move and where capture can be guaranteed in i cop moves after the robber has taken his turn, i.e. for all moves the

robber can take, the resulting state is in W_i.

$$W'_{i+1} = W'_i \cup \{\, \mathbf{s}' \mid \forall \mathbf{s}, (\mathbf{s}', \mathbf{s}) \in E(M) : \mathbf{s} \in W_i \}.$$

Then W_{i+1} is the set of states where capture can occur in i moves plus the states that can be transferred into states in W'_{i+1}.

$$W_{i+1} = W_i \cup \{\, \mathbf{s} \mid \exists \mathbf{s}' : (\mathbf{s}, \mathbf{s}') \in E(M) \wedge \mathbf{s}' \in W'_{i+1} \}.$$

Alternatively, these sets can be thought of as databases of winning states for the cops and the construction of subsequent sets as retrograde analysis [26, 80]. The following theorem can be followed from the known results connected to endgame databases research that any state that is not assigned to any of the winning state databases is a draw. Here, a draw means that the cops cannot enforce capture and therefore that the state is a win for the robber. However, we will outline the mathematical proof for completeness.

Theorem 2.3

Either the cops have or the robber has a winning strategy.

Proof. Since $W_0 \subseteq \ldots \subseteq W_i \subseteq W_{i+1} \subseteq \ldots \subseteq S_C$ and $|S_C| = \frac{1}{2}|S| < \infty$ it follows that there is a j such that $W_j = W_{j+1} = W_m$ for all $m > j$. If $W_j = S_C$, then the cops player has a winning strategy since he can enforce capture in at most j moves. Being in state $\mathbf{s} \in W_i$ he simply moves to a state $\mathbf{s}' \in W'_i$. Since G is connected, such a state $\mathbf{s}'$ exists due to the definition of the W's.
If $W_j \subsetneq S_C$, then we can show that the robber can evade capture forever. First, it has to be determined where the robber chooses to start the game after the cops revealed their initial positions $\mathbf{c}$. Assume the robber cannot choose a vertex r such that $(r, \mathbf{c}, 1) \notin W_j$. Then the cops can win starting in any state in S_C since they can first move to $\mathbf{c}$ and then capture the robber from there. This means $S_C = W_j$ which yields a contradiction. Hence, there is a vertex r that the robber can choose such that $\mathbf{s} = (r, \mathbf{c}, 1) \notin W_j = W_{j+1}$. Let him choose this vertex. Afterwards it is the cops' turn. By the definition of the W's and proof by contradiction it follows that for all possible moves of the cops to a state $\mathbf{s}'$, we have that $\mathbf{s}' \notin W'_{j+1}$ because $\mathbf{s} \notin W_{j+1}$. Now, say the cops chose to move to $\mathbf{s}' = (r, \mathbf{c}', 0)$. Similarly, by the definition of the W''s and proof by contradiction we can follow that there is a r' such that $(r', \mathbf{c}', 1) \notin W_j$ since $\mathbf{s}' \notin W'_{j+1}$. Thus, for every possible move of the cops the robber can always choose a response such that the play never enters either W_j or W'_j. Since all the terminal states are in W_j and W'_j he can evade capture forever. □

Definition 2.4 (cop number)

Given a connected graph G, the *cop number*, denoted by $c(G)$, is the minimal number of cops needed to catch a robber on G. The cop number of a disconnected graph is defined as the maximum of its connected components.

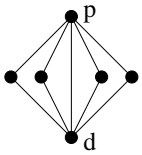


Figure 2.2: Example of a pitfall.

In the following chapters, we will be interested in the time the cops need to capture the robber. Assuming that the cops try to minimize and the robber tries to maximize this time, we will define the search time of a graph and the capture time. The search time has been studied in the literature and we review the known results in Chapter 6. Capture time is a new parameter that has not been studied in the mathematical literature but will be of major importance in the subsequent chapters because the capture time for given initial positions denotes the length of an optimal solution to the game.

Definition 2.5 (search time and capture time)
The *search time*, denoted by $\mathrm{st}(G)$ is the time, i.e. number of (joint) moves, $c(G)$ cops need to catch a robber on G after choosing their initial positions. If the players are not allowed to select their initial positions and play from predefined positions $\mathbf{s}$, we call the time the cops need to capture the robber the *capture time*, denoted by $\mathrm{ct}(\mathbf{s})$. The *capture time of a graph*, denoted by $\mathrm{ct}(G)$ is the maximum of $\mathrm{ct}(\mathbf{s})$ for every state $\mathbf{s}$ in the state space.

Since the search time measures the number of moves taken after the selection of initial positions by both players, it is a property of the graph. In contrast, the capture time with given initial state can vary drastically among the state space. The capture time of the graph measures the worst case the cops can encounter. In Chapter 3 we will investigate algorithms that compute the capture time for every possible state in the state space.

We will now outline the known characterization of 1-cop-win graphs.

Definition 2.6 (neighborhood and pitfall)
Let G be a graph. Then we denote the *neighborhood* of a vertex $v \in V(G)$ as all the vertices that are connected to v, i.e. $N(v) = \{u|(u, v) \in E(G)\}$. Furthermore, $N[v] = N(v) \cup \{v\}$ is referred to as the *closed neighborhood*.
A pair of vertices (p, d) is called a *pitfall* p together with its *dominating vertex* d if $N[p] \subseteq N[d]$.

An example of a pitfall is depicted in Figure 2.2. The underlying idea of a pitfall is that when the cop is in the dominating vertex d and the robber in the dominated vertex p, then no matter where the robber chooses to move, the cop can catch him in his next turn.

Recall that removing a vertex from a graph means deleting the vertex from the vertex set and removing all its incident edges from the edge set of the graph. Aigner and Fromme [1] showed that by deleting any pitfall from a 1-cop-win graph, the obtained graph remains 1-cop-win. They also

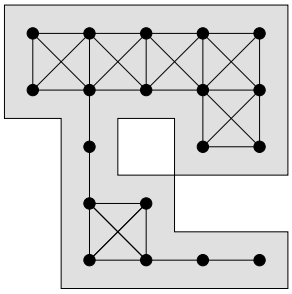


Figure 2.3: A 1-cop-win octile map.

proved the converse, i.e. if the graph obtained from an original graph G by removing a pitfall is 1-cop-win, then G is 1-cop-win. This yields the characterization of 1-cop-win graphs:

Theorem 2.7 (by [1])
Let G be an undirected finite graph. Then G is (1)-cop-win if and only if by successively removing pitfalls (in any order) G can be reduced to a single vertex.

In subsequent chapters, we will conduct experiments on typical game maps and/or mazes. Since we consider the discrete mathematical game, these maps are transferred into graphs. Within this thesis, we will use octile maps for such a conversion. For an example consider the map in Figure 2.3. The resulting graph is a grid with additional diagonal connections which we call octile connected since one vertex can have as many as eight neighbors.

Definition 2.8 (octile map)
Let $G = (V, E)$ be a simple finite undirected graph. G is called an octile map if and only if $V \subset \mathbb{N} \times \mathbb{N}$ and for every vertex $v = (i, j) \in V$

$$
\begin{aligned}
(v, (i, j+1)) \in E &\quad \Leftrightarrow \quad (i, j+1) \in V \\
(v, (i, j-1)) \in E &\quad \Leftrightarrow \quad (i, j-1) \in V \\
(v, (i+1, j)) \in E &\quad \Leftrightarrow \quad (i+1, j) \in V \\
(v, (i-1, j)) \in E &\quad \Leftrightarrow \quad (i-1, j) \in V \\
(v, (i-1, j-1)) \in E &\quad \Leftrightarrow \quad (i-1, j-1), (i-1, j), (i, j-1) \in V \\
(v, (i-1, j+1)) \in E &\quad \Leftrightarrow \quad (i-1, j+1), (i-1, j), (i, j+1) \in V \\
(v, (i+1, j-1)) \in E &\quad \Leftrightarrow \quad (i+1, j-1), (i+1, j), (i, j-1) \in V \\
(v, (i+1, j+1)) \in E &\quad \Leftrightarrow \quad (i+1, j+1), (i+1, j), (i, j+1) \in V
\end{aligned}
$$

and there are no other edges in E.

The definition of an octile map does not include edges for moves over corners of the underlying map. This is due to the fact that we are interested in computer games where characters have a certain width. Hence, they cannot cut through the corners but have to make their way around them maintaining some distance to the wall.

Chapter 3

An optimal algorithm for the full state space

To conquer the enemy without resorting to war is the most desirable. The highest form of generalship is to conquer the enemy by strategy.

Sun Tzu (400 B.C.E. - 320 B.C.E.)

Within this chapter we will explore a solution to compute optimal strategies for the cops and robber game. Here, optimality means playing according the minimax playout. Furthermore, we can compute best responses against a given robber strategy. This computation will be used in Chapter 5 to compare our approximate solutions against optimality and measure how much they can be exploited.

The mathematical literature suggests an algorithm that solves the entire state space [33]. However, no systematic study of the performance of this algorithm has yet been conducted. We consider its theoretical runtime and study its practical performance in Section 3.3. A best response to a fixed robber strategy has been computed by Isaza *et al.* [37] with the help of dynamic programming. We discuss how best responses can be computed efficiently using the optimal algorithm in Section 3.1.

Since this thesis is motivated by computer games, the definition of the cops and robber game, given in Definition 2.1, has to be altered slightly. Rather than allowing players to choose their initial positions, cops and robber are spawned anywhere in the graph. The graph originates from modeling a map and therefore will always be simple, finite, connected and, unless otherwise explicitly mentioned, undirected.

There are different ways to determine when the two players are allowed to take actions. This depends on how time is modeled. There are generally two possibilities. First, time accounts for the number of turns the players have taken. This results in strictly alternating or simultaneous turns. The alternating version is the usual definition of the cops and robber game within the mathematical literature (cf. Definition 2.1) and we will study the simultaneous game in Chapter 7. Second, time measures the distance the agents under each player's control travel, i.e. the time needed to move

along an edge is its edge cost. Then, players might finish their moves at different times during the game. The computation and theory of optimal strategies is severely complicated by allowing a player to announce his next move immediately since a player might move multiple times in a row before the other player takes an action. Furthermore, the move of remaining on the current position becomes continuous since the player might choose any amount of time to sit out. Hence, this possibility will not be investigated in this thesis. Another approach is to make both players wait until they both finish their moves. This is an edge weighted version of the game where time accounts for the number of turns taken. By setting all edge costs to one it becomes an equivalent formulation.

Although all the following algorithms work well with different edge costs, we find that making the players wait until their opponent finishes his turn corrupts the game. For instance, if they have to choose between similar moves (e.g. moves that lead into the same general direction) the robber will always prefer the moves with long edge costs whereas the cop will prefer moves with small edge costs. However, the cop will have to wait until the robber finishes, thus is not rewarded for his choice. Furthermore, in our test environments, optimizing for the traveled distance with real edge costs is equivalent to optimizing the number of turns. Hence, we will set all edge costs to one in our experiments and therefore practically compute the game where time is measured as the number of turns taken.

The *payoff* at the end of the game for the cop player is the negative of the time that passed since the beginning of the game and hence the negation of the time to capture. The payoff for the robber is just the time to capture. Both players want to maximize their payoff, thus the game is a zero-sum two-player game.

It is important to note that we want to compute optimal strategies, i.e. strategies that achieve the minimax value. Therefore, the player can make the assumption that his opponent is playing optimally and does not have to keep track of the history of the game to maximize his payoff. The play is the same when making an (optimal) decision in each turn. In contrast, when assuming that the opponent does not play optimally, the history of the game can be recorded to model the opponent and to increase one's payoff.

Strategies are defined for one player. They assign moves that the player should take to each state where it is the player's turn. Since the game is played alternatingly, it is clear that an optimal strategy exists in pure strategies, hence it suffices to assign only one move to each state.

Definition 3.1 (strategy for the alternating move game)
Let G be the graph that the game is being played on, S its state space and M the move graph as defined in Definition 2.2. Then, a *strategy* for the robber player (the cop player) is a function $\theta : S_R \to S_C$ ($\theta : S_C \to S_R$) such that $(s, \theta(s))$ is an edge in M for every state $s \in S_R$ (S_C).

Definition 3.2 (Nash equilibrium)
Given two strategies θ_C and θ_R for the cop and robber player, respectively, they form a *Nash equilibrium* if no player can increase his payoff by unilaterally changing his strategy. That means by

choosing a different strategy θ'_C, the cop cannot increase his payoff against a robber playing θ_R and the analog holds true for the robber.

Definition 3.3 (game value)
The *value* of a game starting in a state $\mathbf{s}$, denoted by $GV(\mathbf{s})$, is the payoff for the robber when both players play strategies according to a Nash equilibrium.

In alternating games, the value of the game is also referred to as the minimax value of the game. Furthermore, the minimax playout is the playout defined by the Nash equilibrium strategies. We prefer to use these terms in the following sections.

3.1 Retrograde Analysis

Optimal strategies can be generated by computing the minimax value for each state in the state space. Then, being in a state $s_C \in S_C$ the cop chooses a move

$$\underset{s_R \in S_R, (s_C, s_R) \in E(M)}{\arg\min} GV(s_R),$$

that maximally decreases the minimax value of the rest of the game. Analogously, the robber will choose to move to a state that minimally decreases the minimax value. The values can be computed by retrograde analysis, i.e. by building an endgame database.

Retrograde analysis (RA) has been used in other domains, e.g. checkers [71]. It was first suggested for the game of cops and robber by Hahn and McGillivray [33]. Their algorithm initially assigns a value of 0 to each terminal state. Recall that a terminal state is a state $(r, \mathbf{c}, t)$ where the robber is caught, hence $\exists i, 1 \leq i \leq k : c_i = r$. Then, it loops over the entire state space setting the values of states. A state where it is the cops' turn is assigned a value when any of its successors have an assigned value. It is assigned the minimum of the successors' values. A state where it is the robber's turn is assigned a value when all its successors have an assigned value. It is assigned the maximum of the successors' values. The loop repeats until no values change. Note that this is analog to computing the hierarchy of sets W's from Chapter 2, i.e. computing all the positions where the cops win in at most i moves for increasing i.

The algorithm has been rewritten by Thériault [79], Gavenčiak [27] and Isaza [36] to a best first search version. Consider Figure 3.1 for an illustration. The algorithm works with a priority queue. First, all terminal states are assigned a value of 0. Therefore, d, g and p are assigned a value of 0. Then, their predecessors, b, c and k, are scheduled in the queue at the next time step. Afterwards, states will subsequently be taken from the queue ordered by the time they are scheduled. When a state is taken from the queue it is expanded, i.e. its predecessors are generated and considered at the next time step.

States where the cop player, i.e. the minimizing player, is to move can be scheduled immediately. This is the best first search element of the algorithm. States where the robber, i.e. the maximizing

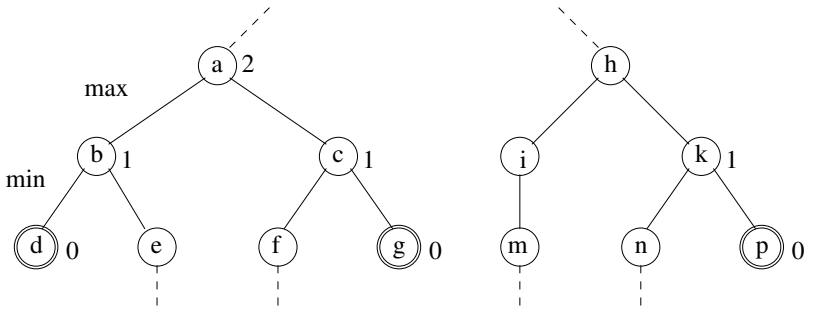


Figure 3.1: Illustration of the RA algorithm.

player, is to move will be delayed until all their children are assigned a value. This is the minimax element of the algorithm. Hence, when b is expanded with a value of 1, a will not be scheduled immediately since c has no assigned value at that moment. Afterwards, when c is expanded with a value of 1, all the children of a are set and therefore, a will be scheduled with a value of 2. Analogously, h will not be scheduled until i is assigned a value.

For a pseudocode implementation consider Figure 3.2. Here, $c(\mathbf{s}, \mathbf{s}')$ denotes the cost of moving from $\mathbf{s}$ to $\mathbf{s}'$, hence the edge cost in the move graph. Furthermore, $N(\mathbf{s})$ is the neighborhood of $\mathbf{s}$ in the move graph. Note that the function compute_max_value has the goal to determine if all the children of a node where the robber is to move already have an assigned value.

In our model, multiple cops can occupy the same vertex. Hence, the size of the state space is bounded by $|S| \leq 2n^{k+1}$ where k is the number of cops that is being played with and n the number of vertices in the graph. Since the game value will only be set once for each state the algorithm clearly terminates.

Note that the given bound for the state space can be reduced by not differentiating between the cops. However, within this section we will not make this distinction to ease reading. The exact bounds will be outlined in Section 3.2.

We will analyze the runtime of the algorithm in the following. There are at most $2kn^k$ terminal states, the k cops can be anywhere in the graph, the robber has to be under one of them and it can be both player's turn. For a given state $\mathbf{s}$ in which the cops are to move, there are at most $(\Delta + 1)^k$ neighbors, where Δ is the maximum degree of the input graph. This is because every cop can potentially move to Δ other vertices or remain on its current position. When the robber is to move in $\mathbf{s}$, then there can be as many as $(\Delta + 1)$ neighbors.

Since we explore the application of the algorithm to game maps we can make the assumption that path costs differ only by a fixed set of values. That means that buckets can be used within the priority queue and that access to the queue can be realized in constant time. Without this assumption, each access to the queue can only be done in logarithmic time.

Under this first game map assumption we thus have that set_terminal_states_values runs in $O(2kn^k(\Delta+1)^k)$. Note that we indirectly assume here that all terminal states can be explicitly generated. This

```
    procedure retrograde_analysis()
1       set_terminal_states_values();
2       while openqueue not empty
3         pop node from openqueue with state s and smallest gcost g;
4         if GV(s) is not set
5           GV(s) ← g;

6           if turn(s) = 1 // cop's turn
7             for all z ∈ N(s)
8               newg ← compute_max_value( z );
9               if GV(z) is not set and newg≠ ∞
10                push a node on openqueue with state z and gcost newg;

11          else // robber's turn
12            for all z ∈ N(s)
13              if GV(z) is not set
14                push a node on openqueue with state z and gcost c(z, s) + GV(s);

    procedure set_terminal_states_values()
15      for all terminal states s ∈ S
16        GV(s) ← 0;
17        if turn(s) = 0 // robber's turn, last move by cops
18          for all z ∈ N(s)
19            push a node on openqueue with state z and gcost c(z, s);

    function compute_max_value( s )
20      r ← −∞;
21      for all z ∈ N(s)
22        if GV(z) is not set return ∞;
23        r ← max( r, c(s, z) + GV(z) );
24      return r;
```

Figure 3.2: Retrograde analysis pseudocode.

might not be true for other domains than cops and robber. The function compute_max_value is only called for states where the robber is to move. Hence it runs in $O(\Delta + 1)$ because of the for loop in Line 21, every other operation can be done in constant time. The while loop in Line 2 is executed at most $O(2n^{k+1})$ times (see above). There are $O((\Delta+1)^k)$ neighbors in Line 7, each for which Line 8 takes $O(\Delta + 1)$ time and there are $O(\Delta + 1)$ neighbors in Line 12. Hence, the entire algorithm runs in

$$O(2kn^k(\Delta + 1)^k + 2n^{k+1}((\Delta + 1)^k(\Delta + 1) + (\Delta + 1))) = O(n^{k+1}(\Delta + 1)^{k+1})$$

since trivially $k \leq n$.

For game maps it is a practical assumption that Δ is bounded above by a constant. This is because game maps are connected in a local manner. Under this second game map assumption, the runtime then becomes $O(n^{k+1}(\Delta + 1)^k)$.

The correctness of the algorithm can be followed by noticing the analogy to the construction of the hierarchy of W's in Chapter 2. In Chapter 4 we enhance this algorithm with a heuristic. Hence, the proof can also be followed from the proof of correctness in Chapter 4 by using the admissible heuristic of zero everywhere.

This algorithm can also be used to compute solutions for directed graphs. In this case, Line 7, 12 and 18 (Figure 3.2) have to be changed such that $N(\mathbf{s})$ denotes the predecessors of $\mathbf{s}$ rather than the successors. On undirected graphs, these two sets are the same.

Note that since the robber is allowed to pass his turn, the cops always take the last move when defeating an optimal robber. This is why we check if it is the robber's turn in the terminal state in Line 17 (Figure 3.2). If the robber is not allowed to pass, he might be forced to run into the cops. That means additionally to Line 17 we would have to extend set_terminal_states_values by

```
else // cop's turn, last move by robber
  for all z ∈ N(s)
    newg← compute_max_value( z );
    if newg≠ ∞
      push a node on openqueue with state z and gcost newg;
```

Furthermore, the algorithm can be used to compute a best response against a robber strategy. Let θ be a robber strategy that does not track the history of the game. A *best response* is a strategy $\bar{\theta}$ for the cop such that the cop maximizes his payoff with $\bar{\theta}$ when facing a robber playing θ. To compute a best response the neighbor relation in Lines 12 and 21 (Figure 3.2) has to be modified. In Line 12 only neighbors from which the robber strategy θ chooses to move into $\mathbf{s}$ have to be considered. In Line 21, only the move that θ chooses has to be assessed.

Since the algorithm works bottom-up it is not possible to compute a best response against target algorithms that use the history of the game. This is due to the fact that the history encodes the play starting in the initial state. Within a bottom-up computation it is not possible to determine how the agents arrived in the positions that are currently evaluated and therefore this information is not available. Thus, it cannot be determined how the target is going to play in Lines 12 and 21.

Analogously, it is possible to compute a best response against a strategy of the cops that does not depend on the history of the game. However, if the cops' strategy has flaws the robber might be able to evade capture forever. States from which infinite escape is possible will not be expanded for the robber since not all their successors will have an assigned value. Thus, the bottom-up computation stalls at these states and no value will be assigned. If the game begins in a state that does not have an assigned value the robber can escape capture forever by choosing moves to non-assigned states. In contrast, if the game starts in states that have an assigned value the robber will get caught within this assigned solution time.

3.2 Optimizations for multiple cops

Recall that the size of the state space is $2n^{k+1}$ since every agent can be on any position in the graph and it can be the cops' or the robber's turn. Therefore, solving the entire state space becomes infeasible very quickly when n or k increase. To reduce the size of the state space notice that we distinguish the positions of all the cops in the state space. However, the cops themselves do not have to be distinguished. Hence, it is possible to reduce the state space size by only considering ordered k-tuples as positions for the cops. Recall further that we allow multiple cops to occupy the same vertex.

Lemma 3.4 *When the cops are not distinguished the game can be encoded with* $2n\binom{n+k-1}{k}$ *states. The number of terminal states is* $2n$ *if* k *equals one and* $2n\binom{n+k-2}{k-1}$ *otherwise.*

Proof. The positions of the k cops can be thought of as combinations with repetition since we do not distinguish the cops. Hence, the number of positions of k cops is $\binom{n+k-1}{k}$. Since the robber can be on any of the n vertices and it can be the cops' or the robber's turn, all the positions of the cops and the robber can be encoded in $2n\binom{n+k-1}{k}$ states. This proves the first part of the Lemma.

Terminal states are states that the robber is caught in. If $k = 1$, i.e. there is only one cop, the robber can be anywhere in the graph with the cop on the same vertex. Hence there are $2n$ terminal states since it can be both player's turn. If $k > 1$, the robber can be on any of the n vertices with one cop on him and the other $k - 1$ cops can be anywhere else in the graph. Since we do not distinguish the other cops there are $\binom{n+(k-1)-1}{k-1}$ possibilities for their positions. Hence the total number of terminal states is $2n\binom{n+k-2}{k-1}$. □

Given the previous algorithm it is important to have a ranking function that maps a given position $(r, c_1, \ldots, c_k)$ bijectively to a number out of $\{0, \ldots, n\binom{n+k-1}{k} - 1\}$. This can be done in the following way. First, transform each position into the number of the respective vertex. Numbering begins at 0 and ends at $(n-1)$. Then, order the numbers of the cops and obtain $(x, y_1, \ldots, y_k)$ where x is the number of the vertex occupied by the robber and $y_1, \ldots, y_k$ are the ordered numbers of the cops' vertices. Using the lexicographical ordering of all possible tuples determines the ranking.

```
function unrank( n, k, ranking )
  x = ⌊ranking / L(n,k)⌋;
  rest = (x + 1)L(n, k) − ranking − 1;
  for j = 1, ..., k
    y_j = 0;
    while L(n − y_j − 1, k − j + 1) > rest
      y_j = y_j + 1;
    rest = rest − L(n − y_j − 1, k − j + 1);
  return (x, y_1, ..., y_k)
```

Figure 3.3: Outline of a function that iteratively constructs the position given its ranking.

In the following, let $L(n,k) = \binom{n+k-1}{k}$ denote the number of possible positions of k cops on n vertices (combinations with repetition). We want to count the number of tuples that would appear earlier in a lexicographical ordering of all possible tuples. Therefore, it is easy to include the robber's position by $xL(n,k)$. Now, consider y_1. When y_1 is set to i there is $(n-i)$ vertices available for all subsequent positions $y_2, \ldots, y_k$, hence $L(n-i, k-1)$ possibilities. Consider y_2. Due to the ordering y_2 has to be at least y_1 and when set to i there are, analog to y_1, $(n-i)$ vertices available for $(k-2)$ more positions. This can be used to count the tuples that appear earlier in a lexicographical ordering by

$$\begin{aligned} H((x, y_1, \ldots, y_k)) &= xL(n,k) + \sum_{i=0}^{y_1-1} L(n-i, k-1) + \ldots + \\ &\quad \sum_{i=y_{j-1}}^{y_j-1} L(n-i, k-j) + \ldots + \sum_{i=y_{k-1}}^{y_k-1} L(n-i, 0), \end{aligned}$$

where H denotes the ranking of the position. Simplifying this expression yields

$$H((x, y_1, \ldots, y_k)) = (x+1)L(n,k) - 1 - \sum_{j=1}^{k} L(n - y_j - 1, k - j + 1). \tag{3.1}$$

Note that this can also be interpreted in a different way. Take the ranking of tuple $(x+1, 0, \ldots, 0)$ which is $(x+1)L(n,k)$ and subtract the number of tuples that come between $(x, y_1, \ldots, y_k)$ and $(x+1, 0, \ldots, 0)$ in a lexicographical ordering. Since we count the number of tuples before the current one we have to subtract one more, hence the -1.

An unranking function that takes a ranking as input and outputs the ordered state $(x, y_1, \ldots, y_k)$ is easy to achieve from equation (3.1). An outline in pseudo code can be found in Figure 3.3. The function determines all the y_j iteratively. It seems unlikely that an explicit formulation can be gained for the general case since the powers of n in equation (3.1) increase with k. Therefore, reversing H gets more and more difficult with increasing k.

However, for the case $k = 1$ and $k = 2$ we can give explicit formulations. Let $k = 1$, then

$$\begin{aligned} x &= \left\lfloor \frac{\text{ranking}}{n} \right\rfloor \\ y &= \text{ranking} - nx. \end{aligned}$$

Let $k = 2$, then

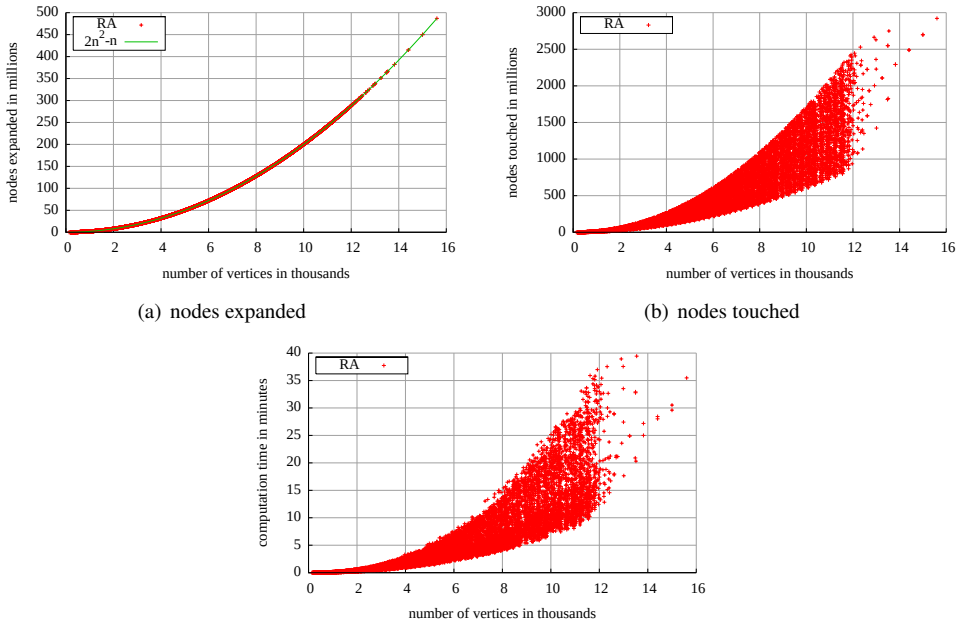
$$\begin{aligned} x &= \left\lfloor \frac{2 \text{ ranking}}{n(n+1)} \right\rfloor \\ y_1 &= \left\lfloor n + \frac{1}{2} - \sqrt{\left(n + \frac{1}{2}\right)^2 - 2\left(\text{ranking} - \frac{n(n+1)}{2}x\right)} \right\rfloor \\ y_2 &= \text{ranking} - \frac{n(n+1)}{2}x - \frac{y_1(y_1+1)}{2} - y_1(n - y_1 - 1). \end{aligned}$$

3.3 Experiments

Within this section, we would like to address the scalability of the retrograde analysis (RA) algorithm from Section 3.1. Although we know the theoretical runtime, it is desired to conduct an empirical study and to answer the question of how large a map can be solved using the RA method and how well the method scales in practice.

We study three different sets of octile connected maps. The first set consists of maps where one cop is sufficient to capture a robber. Since both agents move at the same speed this means that these maps have no obstacles in them, otherwise the robber could loop around them forever. We test on these maps for two reasons. First, we hope to be able to solve large maps since we only use one cop and the state space is therefore manageable. Second, these maps are mazes with potentially big open areas. We can generate this type of maps efficiently. We generated 10,433 maps in the range of 133 to 15,604 vertices and measured the number of node expansions, nodes touched and the required computation time to solve these maps. The results are plotted in Figure 3.4. Note the different scales on the ordinate. The graph in Figure 3.4(a) is a quadratic curve since there are $2n^2$ states and except for the n terminal states in which the cop is to move all of them are expanded exactly once, hence $2n^2 - n$. We only included this plot for verification purposes. In contrast, the number of node expansions can vary significantly due to the structure of the map (see Figure 3.4(b)). Since the algorithm touches in the order of up to 8 times as many nodes as it expands (due to octile connections), the number of nodes touched has the greatest impact on the runtime, depicted in Figure 3.4(c).

To answer the question of how big a map we can actually solve for more interesting cases we will study maps from the commercial game Baldur's Gate in the following. The second set of maps consists of 115 maps, the smallest having 175 and the largest having 22,841 vertices. We solve the game for one robber and one cop where the cop can travel a distance of up to two in every move. Hence, the cop is twice as fast as the robber and the game is finite despite the obstacles in the map. We solve this game with the RA algorithm and measure the number of node expansions, nodes touched and the required computation time. The results are depicted in Figure 3.5 (note the different scales on the ordinate). As a verification we have plotted $2n^2 - n$ in Figure 3.5(a) since there are $2n^2$ states in the state space and n are not expanded in set_terminal_states_values. Note



(a) nodes expanded

(b) nodes touched

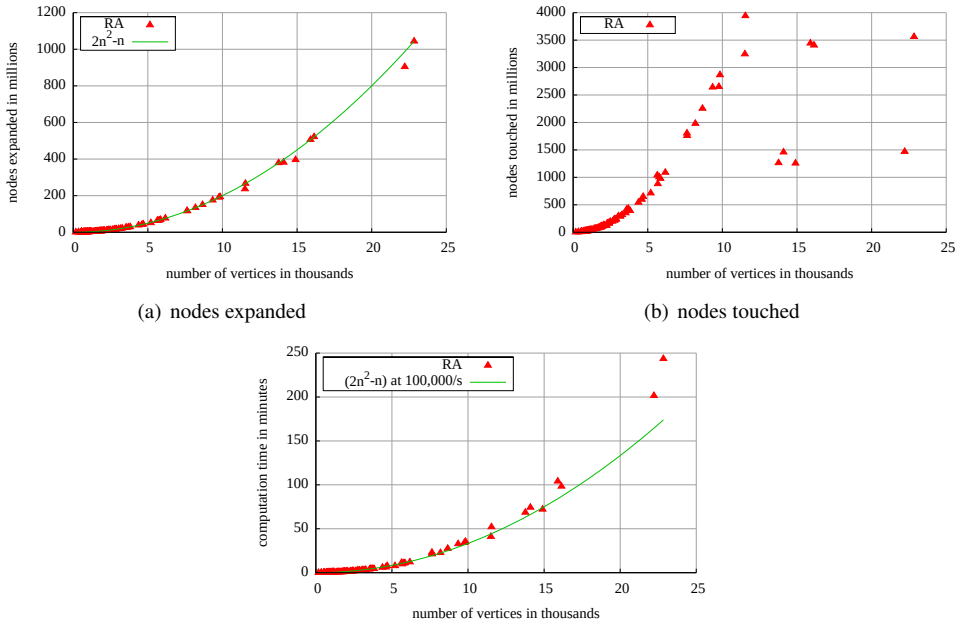
(c) computation time

Figure 3.4: Experiments on 1-cop-win octile connected graphs.

however, that the graph does not perfectly match with our computed values. This is due to the fact that the graphs induced by the Baldur's Game maps are not all connected. Hence, there are initial positions for which the game becomes infinite (e.g. cop and robber in different components). These states are not going to be expanded by the RA algorithm which explains why fewer nodes are getting expanded. For the larger maps, the number of nodes touched varies drastically (Figure 3.5(b)) which has a direct influence on the runtime (Figure 3.5(c)). However, since touching nodes does not take much time compared to expanding a node and pushing it onto the queue, the runtime is proportional to the number of nodes expanded.

The above studies show that the RA algorithm can be used to compute optimal solutions effectively even for large game maps when only one cop is considered. This is mainly due to the quadratic increase in runtime with respect to the number of vertices in the underlying graph. However, as can be seen from Figures 3.4(c) and 3.5(c), the algorithm is far from running in realtime. Hence, its purpose is to compute optimal strategies and best responses offline to be used for assessments of online approximation algorithms (cf. Chapter 5).

We also performed a study for the case when two cops and one robber play and all agents have the same speed. Due to the exponential increase in the size of the state space due to the number of cops, this study was conducted on a third set of smaller maps. On these maps, two cops are sufficient to catch an evading robber. We used 25 maps from the commercial game Baldur's Gate. The smallest map had 175 and the largest 1081 vertices. The state space is of size $n^2(n+1)$ when using two cops. The number of terminal states where the cops are to move is n^2. Since



(a) nodes expanded

(b) nodes touched

(c) computation time

Figure 3.5: Experiments on 115 commercial maps for one fast cop versus one robber.

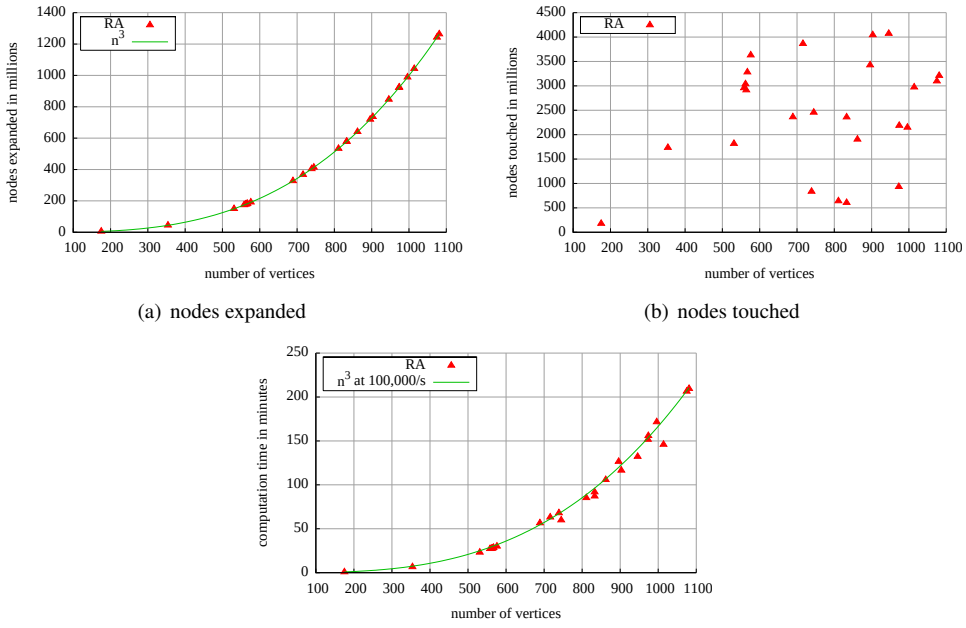
these terminal states are not expanded (in set_terminal_states_values, Figure 3.2) we know that the number of node expansions is $n^2(n+1) - n^2 = n^3$. This is verified in Figure 3.6(a). Again, the number of nodes touched varies drastically due to the different map structures (see Figure 3.6(b)). However, it is remarkable that in the majority of the test runs not more than 5 times as many nodes are touched as expanded although we used two cops and therefore the branching factor is much higher. Furthermore, the highly varying number of nodes touched does not have a big influence on the runtime of the algorithm (Figure 3.6(c)). Theoretically assuming that touching a node does not cost any time, which is far from practice, we can estimate a rate of node expansions of about 100,000 per second.

This study shows that the RA algorithm becomes intractable the more cops are used. The exponential growth of the state space with respect to the number of cops cannot be overcome and it is only possible to compute optimal solutions on relatively small maps, despite all optimization efforts. As a consequence, it is not possible to calculate optimal strategies or best responses that can be used to evaluate online approximation algorithms when computing on large maps with multiple cops.

3.4 Summary

We have defined the basic terms payoff, strategy and optimal value of the game that are needed for game theoretic analysis. Furthermore, we have applied retrograde analysis (RA) for computing these terms.

Although the algorithm is known, its runtime has never been analyzed for the cops and robber



(a) nodes expanded

(b) nodes touched

(c) computation time

Figure 3.6: Experiments on 2-cop-win graphs.

problem and its practical performance was unknown. We studied its theoretical runtime and outlined methods to reduce the state space size to speed up computation. Using these optimizations we conducted experiments on three different sets of maps that show that the RA algorithm can effectively compute optimal solutions for practical problems. However, due to the exponential growth of the state space with respect to the number of cops it is only a feasible approach for up to two cops.

Chapter 4

Optimal algorithms solving single instances

Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest.

Isaac Asimov (1920 - 1992)

In computer games, all agents spawn at their initial positions and play until capture occurs. Therefore, it is desirable to be able to solve the game for one given initial position. The goal is to compute optimal solutions quickly. Hence, it is not practical to solve for the entire state space, as in the previous chapter, and to extract only the solution for one given state.

No study of algorithms that compute optimal playouts or the value of the game for a given initial position has yet been conducted for the game of cops and robber. In fact, the artificial intelligence literature only contains various studies of learning and any-time algorithms that yield good solutions but are not guaranteed to return optimal solutions. None of these studies compare their methods to optimality, in the sense of the minimax playout.

Within the following chapters accumulated edge costs of the moves of all agents are used to determine the payoff that the robber tries to maximize and the cops try to minimize. Within the experiments, we set the edge costs to one to enable effective transposition table lookups for the minimax algorithm (cf. Section 4.1) that we use as a baseline for performance measures. However, all the algorithms that will be described in the following work with any non-negative edge costs.

There are generally two approaches to computing optimal solutions. First, computation beginning with the initial positions and working towards the terminal states, i.e. top-down. Second, computation starting with terminal states and working up in the game tree towards the initial position, i.e. bottom-up. In both situations, a heuristic function can be used to guide the computation. Within top-down methods, the heuristic is used to estimate the value of the game starting in the current state. That means it is an estimate of the distance from the current state to the set of terminal states through optimal play. In contrast, with bottom-up approaches the heuristic function estimates

the distance from the current state to the root of the game tree, i.e. the initial position. This means it estimates if the current position could potentially be reached by optimal play from the root node and if so, what the cost of such a play would be. To avoid confusion we will call a heuristic that is used in a top-down method a *forward heuristic* and a heuristic that is used in a bottom-up approach a *backward heuristic*, unless clear from context.

Inspired by the single agent notations of consistency and admissibility of heuristics we can also define these terms for the two player adversarial game. In single agent search, admissibility of an heuristic means that the heuristic returns at most the exact solution, hence is an underestimation. This can be directly transferred to the two player case. Consistency means that the heuristic of two states does not change more than the transition cost in between the two states. Let h be the heuristic and c be the transition costs. Then, $|h(\mathbf{s}) - h(\mathbf{s}')| \leq c(\mathbf{s}, \mathbf{s}')$ for two adjacent states $\mathbf{s}$ and $\mathbf{s}'$. This can also be interpreted such that a lookahead search of depth one with heuristic leaf node evaluation yields at least the value of the heuristic, i.e. $h(\mathbf{s}) \leq c(\mathbf{s}, \mathbf{s}') + h(\mathbf{s}')$. In the two player case, consistency is only defined for forward heuristics and the single player definition has to be modified to be compatible with the minimax character of the game. Here, consistency means that $h(\mathbf{s}) \leq \min_{\mathbf{s}' \in N(\mathbf{s})}(c(\mathbf{s}, \mathbf{s}') + h(\mathbf{s}'))$ in a state where the minimizer is to move and $h(\mathbf{s}) \leq \max_{\mathbf{s}' \in N(\mathbf{s})}(c(\mathbf{s}, \mathbf{s}') + h(\mathbf{s}'))$ in a state where the maximizer is to move. Hence, a minimax lookahead with depth one and heuristic leaf node evaluation returns at least the heuristic of the root. Note that we will define consistency of a forward heuristic differently in Definition 4.3 but Lemma 4.4 yields the above alternative explanation.

We will first outline the definitions for forward heuristics. Therefore, we define the value of a search to a certain depth where the leaf nodes are evaluated by a given heuristic.

Definition 4.1

Let terminalcost($\mathbf{s}$) denote the cost that is assigned to a leaf node when it is a terminal state. Let $c(\mathbf{s}, \mathbf{s}')$ be the transition cost from state $\mathbf{s}$ to $\mathbf{s}'$. Let d be a given depth and h be a forward heuristic. Then, we define

$$h^*(\mathbf{s}, d) = \begin{cases} \text{terminalcost}(\mathbf{s}) & \mathbf{s} \text{ is a terminal} \\ h(\mathbf{s}) & d = 0 \\ \min_{\text{successors } \mathbf{s}' \text{ of } \mathbf{s}}\{c(\mathbf{s}, \mathbf{s}') + h^*(\mathbf{s}', d-1)\} & \text{Min's turn} \\ \max_{\text{successors } \mathbf{s}' \text{ of } \mathbf{s}}\{c(\mathbf{s}, \mathbf{s}') + h^*(\mathbf{s}', d-1)\} & \text{otherwise} \end{cases}$$

to be the *two agent objective function* computed in state $\mathbf{s}$ to depth d. Furthermore, we say

$$h^*(\mathbf{s}) = \sup_{d \in \mathbb{N}} h^*(\mathbf{s}, d)$$

is the *solution value* for $\mathbf{s}$. Thus, if an optimal solution in $\mathbf{s}$ consists of finitely many moves, we also have $h^*(\mathbf{s}) = h^*(\mathbf{s}, d)$ for sufficiently large d.

For the cops and robber problem, we set all terminal costs to zero. Furthermore, the transition cost $c(\mathbf{s}, \mathbf{s}')$ is the edge cost of the move that one of the players takes to transfer $\mathbf{s}$ into $\mathbf{s}'$.

Definition 4.2 (admissibility and consistency of an algorithm)
Let $\mathbf{s}$ be an initial state. Then, an algorithm is called *admissible* if it computes $h^*(\mathbf{s})$. An algorithm is called *consistent* if it computes $h^*(\mathbf{s}, d)$ when given a depth d as parameter.

Definition 4.3 (admissibility and consistency of a forward heuristic)
A forward heuristic $h : S \to \mathbb{R}$ is called *admissible* if $h(\mathbf{s}) \leq h^*(\mathbf{s})$ for all $\mathbf{s} \in S$. h is called *consistent* if $h(\mathbf{s}) \leq h^*(\mathbf{s}, d)$ for all $\mathbf{s} \in S$ and $d \in \mathbb{N}$.

Analogously to the single agent case it follows immediately from the above definition that consistency of a forward heuristic implies its admissibility.

Lemma 4.4
Let h be a consistent forward heuristic and the transition costs $c \geq 0$. Then, for any $\mathbf{s} \in S$ and any $d \in \mathbb{N}$

$$h(\mathbf{s}) = h^*(\mathbf{s}, 0) \leq h^*(\mathbf{s}, 1) \leq \ldots \leq h^*(\mathbf{s}, d-1) \leq h^*(\mathbf{s}, d).$$

Therefore, the deeper the search in the game tree, the higher the possible values assigned to the root.

Proof. The proof is by induction on d. For $d = 0$, $h(\mathbf{s}) = h^*(\mathbf{s}, 0) \leq h^*(\mathbf{s}, 1)$ for all $\mathbf{s} \in S$ follows by the definition of consistency. Suppose the claim is true for $d = d_0$. If $\mathbf{s}$ is a terminal state, then clearly $h^*(\mathbf{s}, d_0) = h^*(\mathbf{s}, d_0 + 1)$ and the claim holds. Let $\mathbf{z}_1, \ldots, \mathbf{z}_q$ be the successors of $\mathbf{s}$ in the game tree. Then we have by induction hypothesis that $h^*(\mathbf{z}_i, d_0 - 1) \leq h^*(\mathbf{z}_i, d_0)$. Thus, we have

$$\begin{aligned} h^*(\mathbf{s}, d_0) &= \begin{cases} \min_i\{c(\mathbf{s}, \mathbf{z}_i) + h^*(\mathbf{z}_i, d_0 - 1)\} & \text{Min's turn} \\ \max_i\{c(\mathbf{s}, \mathbf{z}_i) + h^*(\mathbf{z}_i, d_0 - 1)\} & \text{otherwise} \end{cases} \\ &\leq \begin{cases} \min_i\{c(\mathbf{s}, \mathbf{z}_i) + h^*(\mathbf{z}_i, d_0)\} & \text{Min's turn} \\ \max_i\{c(\mathbf{s}, \mathbf{z}_i) + h^*(\mathbf{z}_i, d_0)\} & \text{otherwise} \end{cases} \\ &= h^*(\mathbf{s}, d_0 + 1). \end{aligned}$$

□

Definition 4.5 (admissibility of a backward heuristic)
A Nash equilibrium defines a playout starting in an initial position $\mathbf{g} \in S$ (the goal). This playout defines an *optimal path* of moves and both players have no incentive to deviate from it. If the playout of any Nash equilibrium visits state $\mathbf{s}$ we say there is an optimal path from $\mathbf{g}$ to $\mathbf{s}$. Then, we define the cost to get from the initial position $\mathbf{g}$ to a state $\mathbf{s}$ as

$$h^*(\mathbf{g}, \mathbf{s}) = \begin{cases} \text{cost}(p) & \exists \text{ optimal path } p \text{ from } \mathbf{g} \text{ to } \mathbf{s} \\ \infty & \text{otherwise} \end{cases},$$

where cost(p) denotes the accumulated cost of all the moves in p. Note that h^* is well defined since all optimal paths from $\mathbf{g}$ to $\mathbf{s}$ have the same cost. Now, let $h : S \times S \to \mathbb{R}$ be a backward heuristic. We say h is *admissible* if $h(\mathbf{g}, \mathbf{s}) \leq h^*(\mathbf{g}, \mathbf{s})$ for all $\mathbf{g}, \mathbf{s} \in S$.

The algorithms presented in Sections 4.1, 4.2 and 4.3 make use of forward heuristics. In particular, when used with an admissible or consistent heuristic, all these algorithms are admissible or

consistent, respectively. The algorithm developed in Section 4.4 uses backward heuristics and is admissible when used with an admissible heuristic. Note that the term consistency is not defined for backward heuristics.

The immediate approach of using the α-β algorithm [43] will be discussed first. After, we will outline how the optimal algorithm Two-Agent IDA* (TIDA*) [67] can be enhanced with transposition tables to make it feasible for the cops and robber problem. TIDA* uses iterative increase of a lower bound on the solution value. We will show how this bound increase can be incorporated into Proof-Number Search [2], to form the optimal algorithm Iterative Proof-Number Search (IPN). Furthermore, we will use the RA algorithm from Chapter 3 and incorporate heuristic pruning, hence take the transition from an algorithm that works in a best first search manner like Dijkstra's algorithm to an A*-like algorithm. The new algorithm will be called Reverse Minimax A* (RMA*). Last, we conduct experiments with various configurations of heuristics and maps to characterize the performance of the algorithms.

4.1 Minimax

Since the α-β algorithm [43] is a standard algorithm we will omit its precise description. Our implementation computes to a given depth d and evaluates the leaf nodes with a heuristic h if they are not terminal. Within our experiments, we precompute the optimal depth needed to solve the problems with the subsequent algorithms. This depth is then given to the α-β algorithm to measure its performance. If the optimal depth would not be supplied, the algorithm would have to find it by iterative deepening [45], i.e. subsequent searches with increasing depths.

Transposition tables cache the g-cost (pathcost from the root) of a state $\mathbf{s}$. Caching the g-cost is necessary since we use edge costs on the move graph to determine a state's value together with a fixed lookahead. We incorporated heuristic pruning whenever $\beta \leq h(\mathbf{s})$ ($\beta \leq h(\mathbf{s}, d_{\mathbf{s}})$) in an explored node $\mathbf{s}$ (at depth $d_{\mathbf{s}}$). It is easy to see that this does not change the correctness of the algorithm.

Theorem 4.6 (by [43])
When used with an admissible (consistent) forward heuristic, the α-β algorithm is admissible (consistent).

We also investigated another method of pruning. The playout of the game is clearly suboptimal when it revisits the same state. Since we are only interested in optimal solutions here, cycle detection for the currently explored path from the root to the current state can be used for pruning. However, in contrast to pruning with an admissible or consistent heuristic, the algorithm does not remain admissible when using cycle detection. For an example, consider the graph in Figure 4.1. All edge costs are set to one. Here, the robber starts at vertex 5 and the cop at vertex 0, hence in positions $(5, 0)$ with the cop beginning. We prune the search whenever a repetition is encountered and compute to

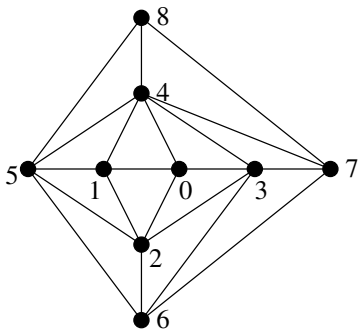


Figure 4.1: Graph to demonstrate the GHI problem.

depth nine with heuristic evaluation of the leaf nodes. The heuristic evaluates the rest of the playout by assuming that the robber stays still and the cop runs towards him on a shortest path. Let the first search be as follows: $(5,0) \overset{\text{cop}}{\rightarrow} (5,1) \overset{\text{robber}}{\rightarrow} (8,1) \overset{\text{cop}}{\rightarrow} (8,4) \overset{\text{robber}}{\rightarrow} (7,4) \overset{\text{cop}}{\rightarrow} (7,3) \overset{\text{robber}}{\rightarrow} (8,3)$. The possible successor state $(8,4)$, i.e. the cop moving to 4, will now be pruned as a repetition. Only the successors $(8,7)$, $(8,0)$, $(8,2)$ and $(8,3)$ remain. Since we search to depth nine these successors are evaluated with a search of a remaining depth of two with the robber beginning. In these searches, the robber can escape and end up at least at distance one from the cop. Hence, the above successors evaluate to a cost of at least four. This is propagated up and a transposition table entry is stored for $(8,3)$ at depth six with a value of five. This is propagated up to $(7,3)$ at depth five. Recall that it is the robber's turn at depth five and therefore the successor with the highest evaluation is choosen. All successors of $(7,3)$ other than $(8,3)$ result in immediate capture or capture after one more move of the cop. Therefore, a transposition table entry is stored for $(7,3)$ at depth five with value six. Now consider a search at a later time: $(5,0) \overset{\text{cop}}{\rightarrow} (5,1) \overset{\text{robber}}{\rightarrow} (6,1) \overset{\text{cop}}{\rightarrow} (6,2) \overset{\text{robber}}{\rightarrow} (7,2) \overset{\text{cop}}{\rightarrow} (7,3)$. Due to the stored transposition table entry, the search is pruned with cost six. Therefore, this search obtaines cost at least eleven. However, the optimal solution is unique except for the choices in the last two moves, has value nine and can only be obtained by this later search: $(5,0) \overset{\text{cop}}{\rightarrow} (5,1) \overset{\text{robber}}{\rightarrow} (6,1) \overset{\text{cop}}{\rightarrow} (6,2) \overset{\text{robber}}{\rightarrow} (7,2) \overset{\text{cop}}{\rightarrow} (7,3) \overset{\text{robber}}{\rightarrow} (8,3) \overset{\text{cop}}{\rightarrow} (8,4) \overset{\text{robber}}{\rightarrow} (8,4) \overset{\text{cop}}{\rightarrow} (8,8)$.

In summary, a repetition in the history of the first search causes a state to be pruned. Its predecessor is stored in the transposition table with a wrong value since the pruned successor is not available. When this predecessor is reencountered in a later search with a different search history, it is pruned incorrectly due to the transposition table entry. Note that we have only depicted an example where a pruned cop move that leads into a repetition causes incorrect computation. However, it is also possible to construct examples where a pruned robber move that leads into a repetition causes incorrect computation.

This problem is known as the Graph History Interaction Problem (GHI). Solutions to this problem exist. Kishimoto and Müller [42] developed a general method to resolve the GHI problem. However, in first experiments we found that using cycle detection does not yield significant speedups. This is due to the interaction of the admissible heuristic pruning with the α-β window. The interval

spanned by α and β contracts with increasing depth of the search. Since the heuristic prunes nodes whenever $\beta \leq h(\mathbf{s})$ and β becomes subsequently smaller, pruning appears at already reasonable depths. Thus, we did not use cycle detection in our implementation and hence also circumvent the GHI problem.

4.2 Two-Agent IDA*

Two-Agent Iterative Deepening A* (TIDA*) has been developed by Prieditis and Fletcher [67], however it is not widely-known. The algorithm is designed for two player adversarial games where players want to minimize or maximize accumulated transition and terminal costs. Heuristic bounds are used to prune the search space in a similar way to how the single-agent IDA* algorithm prunes nodes with high costs. The cops and robber domain has these properties, in that the goal is to maximize/minimize the distance to capture, which can be estimated with a forward heuristic.

An outline of the algorithm can be found in Figure 4.2. Lines 8-12 and 25-28 are the caching enhancements that will be explained later. Line 14 belongs to the artificial depth limit d that is used in the original TIDA* algorithm to ensure consistency. Since we are interested in optimal solutions we can remove this parameter. In the following, we will therefore discuss the algorithm without the depth parameter but all the discussion applies to the consistent case as well.

Analogously to the single-agent case we say the f-cost of a node is the g-cost at which it is encountered together with its heuristic cost. The basic idea is to maintain a lower bound b on the value and f-cost of the root, i.e. $b \leq h^*(\mathbf{s})$ and prove or disprove this bound. Since $h(\mathbf{s}) \leq h^*(\mathbf{s})$ when h is admissible, we can safely assume a first lower bound $h(\mathbf{s})$ (Line 1 of Figure 4.2) and set the current bound to this value (Line 3). This bound is then either proved or disproved by a TIDA* iteration. The iteration has to return a new lower bound on the root's value greater than the current one. If there is no such higher bound, we have found a solution with the cost of the current bound.

Consider the game tree in Figure 4.3 as a sample TIDA* computation. We have not plotted the entire tree but only the important parts. For simplicity, all actions have a cost of one. Furthermore, the minimizing player begins the game. Each circle signifies a state and the number within the circle its heuristic cost. The dotted line marks the part of the game tree that will be expanded by TIDA* in the first iteration. Since the root has a heuristic cost of three, the first iteration is given a bound of three. Afterwards, a cost of five will be propagated up to the root. This is used as the bound for the next iteration. The dashed line (Figure 4.3) marks the part of the game tree that will be expanded by the second iteration.

Within an iteration it is first determined whether $\mathbf{s}$ is a terminal state (Line 7), in which case the robber has been captured and the cost of capture is returned. Recall that the cost of a terminal for the standard cops and robber problem is zero. Next, we ensure that a capture is possible. If the current bound in this iteration is less than what is needed to reach a terminal state, i.e. exceeds the f-cost, the higher heuristic value is returned (Line 13). In the consistent case the algorithm only computes

```
    function tida( s, d )
1       r ← h(s, d);
2       do
3         b ← r;
4         r ← tida_iteration(s, b, d);
5       while r > b;
6       return r;
```

```
    function tida_iteration( s, b, d )
7       if s is a terminal state return terminalcost(s);
8       if upper_bound_cache(s) exists
9         if upper_bound_cache(s) ≤ b return upper_bound_cache(s);
10      if lower_bound_cache(s) exists
11        if b < lower_bound_cache(s) return lower_bound_cache(s);
12      else
13        if b < h(s, d) return h(s, d); // heuristic prune
14      if d = 0 return b; // depth prune

15      if // Min's turn in s
16        r ← ∞;
17        for all z ∈ N(s);
18          r = min( r, c(s, z) + tida_iteration(z, b − c(s, z), d − 1) );
19          if r ≤ b break; // α prune

20      else // Max's turn in s
21        r ← −∞
22        for all z ∈ N(s)
23          r = max( r, c(s, z) + tida_iteration(z, b − c(s, z), d − 1) );
24          if r > b break; // β prune

25      if b < r
26        lower_bound_cache(s) = r;
27      else
28        upper_bound_cache(s) = r;
29      return r;
```

Figure 4.2: Two-Agent IDA* with depth limit and bound caching.

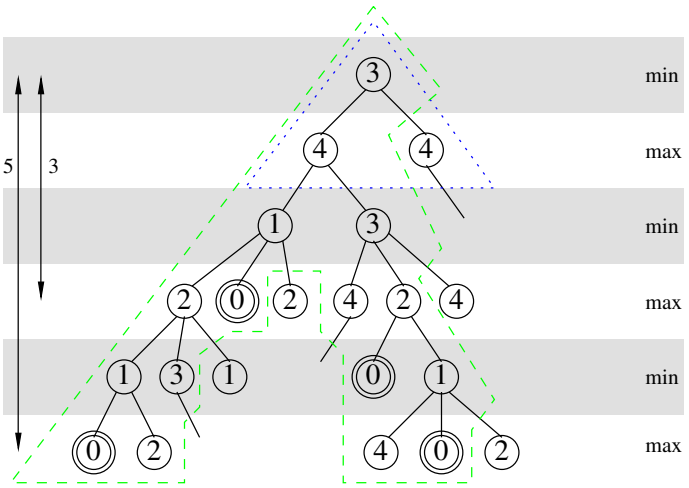


Figure 4.3: Example computation of TIDA* returning a solution value of five. All actions have a cost of one and terminals have a cost of zero. Each circle signifies a state and the number within the circle its heuristic cost. Terminal states are visualized with a doubled circle. The dotted line marks the expanded tree in the first iteration with a bound of three. The dashed line marks the expanded tree in the second iteration with a bound of five.

up to a given depth d. If this depth is reached, the value of the current bound b is returned (Line 14).

Reconsider the example in Figure 4.3. Within the first iteration with a bound of three, the root is expanded and its successors are considered. Both have a heuristic cost of four and are explored at a cost of one. Therefore, their f-cost is five which is greater than the current bound of three. Hence, both nodes are pruned and the search returns to the root with a backed up value of five. The next iteration then starts with this returned value as its bound.

If these pruning cases do not apply, TIDA* performs a typical depth-first search by recursively operating on all the successors $\mathbf{z}$ of $\mathbf{s}$ with a decreased bound $(b - c(\mathbf{s}, \mathbf{z}))$. Lines 19 and 24 do α and β pruning similar to the α-β algorithm [43].

Theorem 4.7 (by [67]) *Given a consistent (forward) heuristic h, we have* $\text{tida}(\mathbf{s}, d) = h^*(\mathbf{s}, d)$.

Theorem 4.8

Let $c(\mathbf{s}, \mathbf{s}') > 0$ for all $\mathbf{s}, \mathbf{s}' \in S$ $(\mathbf{s} \neq \mathbf{s}')$ and $1 < |S| < \infty$. Let further h be an admissible forward heuristic and $h(\mathbf{s}) \geq 0$ for all $\mathbf{s} \in S$. Then $\text{tida}(\mathbf{s}) = h^*(\mathbf{s})$.

Before proving Theorem 4.8 we will prove two lemmata that establish bounds on the outcome of a TIDA* iteration. The lemmata disregard caching (Lines 8-12 and 25-28 in Figure 4.2). In fact, the caching is motivated by these lemmata and we will show that it does not change the outcome of the computation afterwards. Both lemmata are very similar to the consistent case in the original TIDA* paper [67]. However, we review the proofs here to gain necessary insights for the next section.

Lemma 4.9

Let h be an admissible heuristic, $h \geq 0$ and the transition costs $c \geq \epsilon > 0$. Let v be the value returned by $\text{tida\_iteration}(\mathbf{s}, b)$. *If $b < v$ then $v \leq h^*(\mathbf{s})$.*

Proof. In each recursion, the depth bound is decreased by at least ϵ and as soon as $b < 0$ the algorithm prunes the search since $b < 0 \leq h(\mathbf{s})$ for all $\mathbf{s} \in S$. It follows, that the computed part of the game tree by tida_iteration has a maximal depth. Hence, the subroutine terminates.
The rest of the proof is similar to the one given in [67]. The original proof is by induction over the depth. Since we discarded the depth limit parameter our proof is by induction over the bound. We first prove the induction hypothesis. Let $b < 0$.
Case 1: the algorithm returns because $\mathbf{s}$ is a terminal state.

$$v = \text{terminalcost}(\mathbf{s}) = h^*(\mathbf{s})$$

Case 2: the algorithm returns because $b < h(\mathbf{s})$. Since h is admissible we have

$$v = h(\mathbf{s}) \leq h^*(\mathbf{s})$$

There are no other cases because the condition in Case 2 is always true since $h \geq 0$. Now let $b \leq b_0 + \epsilon$ where b_0 is the induction basis. The first two cases are analog to the above since we did not use $b < 0$ within them.
Case 3: it is the minimum player's turn. Since $b < v$ it follows that there was no α-prune. Hence, all the successors have been visited. Therefore, let $\mathbf{z}$ denote the successor such that

$$\mathbf{z} = \operatorname*{arg\,min}_{\mathbf{s}' \in N(\mathbf{s})}(c(\mathbf{s}, \mathbf{s}') + h^*(\mathbf{s}')).$$

By the calculation of v and the fact that $b < v$ we have

$$\begin{aligned} b &< v \leq c(\mathbf{s}, \mathbf{z}) + \text{tida\_iteration}(\mathbf{z}, b - c(\mathbf{s}, \mathbf{z})) \quad \Leftrightarrow \\ b - c(\mathbf{s}, \mathbf{z}) &< \text{tida\_iteration}(\mathbf{z}, b - c(\mathbf{s}, \mathbf{z})). \end{aligned}$$

Since $b - c(\mathbf{s}, \mathbf{z}) \leq b - \epsilon \leq b_0$ we have by induction hypothesis that $\text{tida\_iteration}(\mathbf{z}, b - c(\mathbf{s}, \mathbf{z})) \leq h^*(\mathbf{z})$. Therefore,

$$v \leq c(\mathbf{s}, \mathbf{z}) + \text{tida\_iteration}(\mathbf{z}, b - c(\mathbf{s}, \mathbf{z})) \leq c(\mathbf{s}, \mathbf{z}) + h^*(\mathbf{z}) = h^*(\mathbf{s}).$$

Case 4: it is the maximizer's turn. Since $b < v$ the β-prune took effect. Therefore, let $\mathbf{z}$ be the successor that has been visited last. We have

$$\begin{aligned} b &< v = c(\mathbf{s}, \mathbf{z}) + \text{tida\_iteration}(\mathbf{z}, b - c(\mathbf{s}, \mathbf{z})) \quad \Leftrightarrow \\ b - c(\mathbf{s}, \mathbf{z}) &< \text{tida\_iteration}(\mathbf{z}, b - c(\mathbf{s}, \mathbf{z})) \end{aligned}$$

and since $b - c(\mathbf{s}, \mathbf{z}) \leq b - \epsilon \leq b_0$ we can apply the induction hypothesis. Therefore,

$$v = c(\mathbf{s}, \mathbf{z}) + \text{tida\_iteration}(\mathbf{z}, b - c(\mathbf{s}, \mathbf{z})) \leq c(\mathbf{s}, \mathbf{z}) + h^*(\mathbf{z}) \leq h^*(\mathbf{s}).$$

This completes the proof. □

Lemma 4.10

For the same assumptions as in Lemma 4.9 we have: if $v \leq b$ then $h^*(s) \leq v$.

Proof. We will use the same notation as in Lemma 4.9. Analogously, we can also apply induction here. Therefore, let $b < 0$.

Case 1: the algorithm returns because $\mathbf{s}$ is a terminal state. This is analog to Lemma 4.9.

Case 2: the algorithm returns because $b < h(\mathbf{s})$. This is impossible since $v = h(\mathbf{s}) \leq b < h(\mathbf{s})$.

Note that since for $b < 0$ the condition in Case 2 is always true, execution of tida_iteration must have been ended at Case 1. Now let $b \leq b_0 + \epsilon$ where b_0 is the induction basis. The first two cases are analog to the above.

Case 3: it is the minimizer's turn. Since $v \leq b$ the α-prune took effect. Let $\mathbf{z}$ be the last visited successor of $\mathbf{s}$. Since

$$\begin{aligned} v = c(\mathbf{s},\mathbf{z}) + \text{tida\_iteration}(\mathbf{z}, b - c(\mathbf{s},\mathbf{z})) &\leq b \quad \Leftrightarrow \\ \text{tida\_iteration}(\mathbf{z}, b - c(\mathbf{s},\mathbf{z})) &\leq b - c(\mathbf{s},\mathbf{z}) \end{aligned}$$

and $b - c(s,n) \leq b - \epsilon \leq b_0$ we can apply the induction hypothesis and obtain due to the definition of h^*

$$h^*(\mathbf{s}) \leq c(\mathbf{s},\mathbf{z}) + h^*(\mathbf{z}) \leq c(\mathbf{s},\mathbf{z}) + \text{tida\_iteration}(\mathbf{z}, b - c(\mathbf{s},\mathbf{z})) = v.$$

Case 4: it is the maximizer's turn. Since $v \leq b$ there has been no β-prune. Hence let $\mathbf{z}$ be the child of $\mathbf{s}$ such that

$$\mathbf{z} = \underset{\mathbf{s}' \in N(\mathbf{s})}{\arg\max}(c(\mathbf{s},\mathbf{s}') + h^*(\mathbf{s}')).$$

Since

$$\begin{aligned} c(s,\mathbf{z}) + \text{tida\_iteration}(\mathbf{z}, b - c(\mathbf{s},\mathbf{z})) &\leq v \leq b \quad \Leftrightarrow \\ \text{tida\_iteration}(\mathbf{z}, b - c(\mathbf{s},\mathbf{z})) &\leq b - c(\mathbf{s},\mathbf{z}) \end{aligned}$$

and $b - c(\mathbf{s},\mathbf{z}) \leq b - \epsilon \leq b_0$ we can apply the induction hypothesis. Therefore,

$$h^*(\mathbf{s}) = c(\mathbf{s},\mathbf{z}) + h^*(\mathbf{z}) \leq c(\mathbf{s},\mathbf{z}) + \text{tida\_iteration}(\mathbf{z}, b - c(\mathbf{s},\mathbf{z})) \leq v.$$

This completes the proof. □

Proof of Theorem 4.8. Let $\epsilon = \max_{\mathbf{s},\mathbf{s}' \in S} c(\mathbf{s},\mathbf{s}') > 0$ (note that the maximum exists since $|S| < \infty$). Recall the algorithm. We start with setting the bound $b \leftarrow h(\mathbf{s}) \leq h^*(\mathbf{s})$. By Lemma 4.9 we know that if $b < v$ then $v \leq h^*(\mathbf{s})$ and we increase the bound $b \leftarrow v$. By Lemma 4.10 we know that if $v \leq b$ then $h^*(\mathbf{s}) \leq v$ and we stop iterating. Therefore, after the last iteration we have

$$b \leq h^*(\mathbf{s}) \leq v \leq b.$$

and thus $b = v = h^*(\mathbf{s})$.

It remains to show, that the algorithm terminates, i.e. the values of the ever increasing bounds do not have an accumulation point smaller than $h^*(\mathbf{s})$. In the following, we show that there is only a finite number of values that b can encounter.

From the argumentation at the beginning of the proof of Lemma 4.9 it follows that the height of the tree that TIDA* computes is at most $d = \lfloor h^*(\mathbf{s})/\epsilon \rfloor$. The number of possible path lengths from the root to the bottom of the tree is bounded by

$$1 + |S| + |S|^2 + \ldots + |S|^d = \frac{|S|^{d+1} - 1}{|S| - 1} < \infty.$$

Therefore, the total number of values b can encounter, i.e. the total number of iterations, is bounded above. □

Since the move graph of the cops and robber domain has many cycles (the shortest cycle is length 4: both players taking a move and reversing it), caching or transposition tables are expected to improve performance. A first approach is to cache the outcome of tida_iteration due to submitted state $\mathbf{s}$ and bound b. First experiments show, that this indeed yields reasonable speedups since most states are re-encountered with the same bound. However, more improvements can be made.

Lemmata 4.9 and 4.10 suggest a different caching method. Whenever we return a value $v > b$ we know by Lemma 4.9 that v is a lower bound on the true value. Therefore, we can store this value in a lower bound cache (Line 26 in Figure 4.2). On the other hand, if $v \leq b$, we know by Lemma 4.10 that v is an upper bound on the true value and therefore store v in an upper bound cache (Line 28).

This new method of caching does not alter the admissibility of the algorithm. Suppose an upper bound u for $h^*(\mathbf{s})$ has been cached and tida_iteration is called with bound b such that $u \leq b$. Then $h^*(\mathbf{s}) \leq b$ and it follows that $h^*(\mathbf{s}) \leq \text{tida\_iteration}(\mathbf{s}) \leq b$ (proof by contradiction with the help of Lemma 4.9). Since $h^*(\mathbf{s}) \leq u \leq b$ we can return u immediately in Line 9 because it satisfies Lemma 4.10. Suppose a lower bound ℓ for $h^*(\mathbf{s})$ has been cached and tida_iteration is called with bound b such that $b < \ell$. Then $b < h^*(\mathbf{s})$ and it follows that $b < \text{tida\_iteration}(\mathbf{s}) \leq h^*(\mathbf{s})$ (proof by contradiction with the help of Lemma 4.10). Since $b < \ell \leq h^*(\mathbf{s})$ we can return ℓ in Line 11 because it satisfies Lemma 4.9. This shows that the caching methods obey Lemma 4.9 and 4.10. Hence, by Theorem 4.8 the algorithm with our caching is admissible.

This caching is superior to the simplest technique of caching the value for tida_iteration due to state and bound. This is because the lower and upper bounds get progressively tighter as we search. Due to the checks in Lines 9 and 11 (Figure 4.2), when updating the bounds in Lines 26 and 28 we have either

$$r \leq b < \text{upper\_bound\_cache}(\mathbf{s}) \quad \text{or}$$
$$\text{lower\_bound\_cache}(\mathbf{s}) \leq b < r.$$

```
    function ipn( s )
1       r ← h(s);
2       do
3         b ← r;
4         r ← ipn_iteration(s);
5       while r > b;
6       return r;
```

(a) The algorithm's loop to increase the bound b

```
    function ipn_iteration(s)
1     create a node root with state(root) ← s;
2     while proof_number(root) ≠ 0 and disproof_number ≠ 0
3       node ← select_most_proving_node(root);
4       if node not flagged as being pruned
5         expand_node(node);
6       update_proof_numbers(node);
7     return value(root);
```

(b) Iteration until the root node is proved or disproved

Figure 4.4: Iterative Proof-Number Search main loop and iteration.

When trying to extract solution paths, the α-prune of TIDA* has to be modified. Although we know that the solution is at least b (in tida_iteration), we did not necessarily explore a path with this cost yet. Therefore, Line 19 in Figure 4.2 has to be changed to:

if $r < b$ **break**;

to ensure exploration to the terminal nodes. The paths themselves can easily be stored in the upper bound cache. First experiments indicate that this change results in drastically higher runtimes. Hence, we only report the result with pruning in the case of "$\leq$" in the experiments. If the computed playout that achieves the optimal value is needed, rather than just the value of the game, further analysis is necessary.

TIDA* is related to the algorithm AO*[52] in the sense that the heuristic is used to guide the top-down search. Informally, both algorithms expand the game tree until a next higher heuristic cost is reached (see Figure 4.3 and the parts of the game tree that are marked with dotted and dashed lines for an illustration). AO* only expands the states with new heuristic cost. It does not reexpand the previously computed part of the game tree. These new states are expanded subsequently and therefore the algorithm works like a breadth-first search. To propagate the new information from an expanded state up in the game tree the algorithm updates all the predecessors of the expanded state. Hence, AO* expands the game tree in an A*-like fashion updating the predecessors at every expansion. TIDA* runs a depth-first search to expand all the states of the game tree that do not exceed the current cost bound. In contrast to AO* this search reexpands the previously computed part of the game tree. However, no additional updates of the predecessors are needed since these updates are performed as part of the depth-first search. Hence, TIDA* expands the game-tree in an IDA*-like fashion.

4.3 Iterative Proof-Number Search

The above discussion of TIDA* and its ability to prove or disprove the value of the root suggests a different method than depth-first search (as done by TIDA*). Proof number search (pn-search) was developed by Allis *et al.* [2] to prove or disprove the value of the root given a tree with binary values.

```
    function select_most_proving_node( node )
1     while node is not a terminal node
2       if try_pruning_node( node )
3         return node;
4       if Min's turn in state(node)
5         node ← leftmost child p with disproof_number(p) = disproof_number(node);
6       else // Max's turn
7         node ← leftmost child p with proof_number(p) = proof_number(node);
8     return node;
```

Figure 4.5: Selection of the next node to be expanded.

```
      procedure expand_node( node )
1       for all successors s of state(node)
2         generate new node p;
3         state(p) ← s;
4         parent(p) ← node;
5         bound(p) ← bound(node) - c(state(node), s);
6         if s is a terminal state;
7           value(p) ← terminalcost(s);
8           if bound(p) < value(p) // s has been proved
9             proof_number(p) ← 0;
10            disproof_number(p) ← ∞;
11          else  // s has been disproved
12            proof_number(p) ← ∞;
13            disproof_number(p) ← 0;
14        else // s is not a terminal state
15          if not try_pruning_node( p )
16            value(p) ← bound(p);
17            proof_number(p) ← 1;
18            disproof_number(p) ← 1;
```

Figure 4.6: Expansion of a node at the bottom of the tree.

```
    procedure update_proof_numbers( node )
1     while node exists
2       s ← state(node);

3       if Min's turn in s
4         proof_number(node) ← min_{children p of node} proof_number(p);
5         disproof_number(node) ← ∑_{children p of node} disproof_number(p);
6         value(node) ← min_{children p of n} value(p);

7       else // Max's turn in s
8         proof_number(node) ← ∑_{children p of node} proof_number(p);
9         disproof_number(node) ← min_{children p of node} disproof_number(p);
10        value(node) ← max_{children p of node} value(p);

11      if proof_number(node) = 0
12        lower_bound_cache(s) ← max( lower_bound_cache(s), value(node) );
13      if disproof_number(node) = 0
14        upper_bound_cache(s) ← min( upper_bound_cache(s), value(node) );

15      node ← parent(node);
```

Figure 4.7: Bottom-up recursive update of the nodes in the computed tree.

```
    function try_pruning_node( node )
1     pruned ← false;
2     s ← state(node);
3     if upper_bound_cache(s) exists
4       if upper_bound_cache(s) ≤ bound(node)
5         value(node) ← upper_bound_cache(s);
6         proof_number(node) ← ∞;
7         disproof_number(node) ← 0;
8         pruned ← true;
9     if lower_bound_cache(s) exists
10      if bound(node) < lower_bound_cache(s)
11        value(node) ← lower_bound_cache(s);
12        proof_number(node) ← 0;
13        disproof_number(node) ← ∞;
14        pruned ← true;
15    else // lower_bound_cache(s) does not exist
16      if bound(node) < h(s)
17        value(node) ← h(s);
18        proof_number(node) ← 0;
19        disproof_number(node) ← ∞;
20        pruned ← true;
21    if pruned
22      flag node as being pruned;
23    return pruned;
```

Figure 4.8: The function that tries to prune a node and if successful returns true, false otherwise.

On multi-valued trees, they suggest using pn-search to prove or disprove a lower bound on the root's value. The algorithm keeps proof and disproof numbers for each node in the game tree that indicate how many nodes in the subtree under the node have to be proved to guarantee the current bound (proof number) and how many have to be proved to break the current bound (disproof number). Based on these numbers, in each iteration, the most proving node, i.e. the node with the possibly most impact on the (dis)proof of the root's bound, is selected for expansion. The original algorithm expands nodes until the root's value is (dis)proved and then starts a new iteration with a new lower bound, finding the correct value by binary-search.

Since we want to compute optimal solutions, i.e. create an admissible algorithm, we only look at the admissible case here. In case a consistent algorithm is needed (cf. Definition 4.2), the following can be easily adopted.

In our version of pn-search, we say a state $\mathbf{s}$ is proved for a given bound b if $b < h^*(\mathbf{s})$ and therefore disproved when $b \geq h^*(\mathbf{s})$. Additionally to (dis)proof numbers, we also give every node in the computed tree a value that is assigned analogously to TIDA* and is propagated up towards the root when the proof and disproof numbers are updated. Therefore, the iterative increase of the bound and the pruning techniques known from TIDA* can be used within this algorithm, hence the name *iterative proof-number search* (IPN). A pseudo code implementation can be found in Figures 4.4-4.8. Most of the syntax has been borrowed from Allis *et al.* [2]. However, we outline the entire algorithm to emphasize where changes have been made. The normal pn-search

routine select_most_proving_node has been altered in Lines 2 and 3, expand_node in Line 15 and update_proof_numbers in Lines 11-14. The other routines are new.

Analogous to TIDA*, our main loop iteratively increases the bound until a solution is found (Figure 4.4(a)). The algorithm keeps expanding nodes until the root is either proved or disproved (Figure 4.4(b)) and returns the value of the (dis)proved root to the main loop. When traversing the tree in search of the next node to be expanded, some nodes along the paths might be prunable because of previous updates to the upper and lower bound cache (Line 2 in Figure 4.5). Additionally, when expanding the next node pruning has to be taken into account as well (Line 15 in Figure 4.6). Furthermore, when updating the proof and disproof numbers, the upper and lower bound cache is updated (Lines 11 to 14 in Figure 4.7). The pruning itself works analogously to TIDA* (Figure 4.8).

Theorem 4.11

Under the assumptions of Theorem 4.8 we have: $\text{ipn}(\mathbf{s}) = h^*(\mathbf{s})$.

Proof. We first show, that when ipn_iteration returns with value v for a given state $\mathbf{s}$ and bound b after $\mathbf{s}$ has been (dis)proved, we have

- if $b < v$ then $v \leq h^*(\mathbf{s})$ (similarly to Lemma 4.9) and
- if $b \geq v$ then $v \geq h^*(\mathbf{s})$ (similarly to Lemma 4.10).

Recall that within an iteration we want to prove $b < h^*(\mathbf{s})$ and only end the iteration if either $\mathbf{s}$ has been proved or disproved. Analogously to the proof of Theorem 4.8, the proof is by induction over intervals of b.

Let n be a node that is encountered during the search. The two claims clearly hold if n is a terminal node and if n is pruned by the heuristic (see the proofs of Lemma 4.9 and 4.10 for more details). Since $h \geq 0$ and we prune whenever $b < h(\mathbf{s})$, we showed the induction hypothesis for $b < 0$. In the following let $b \leq b_0 + \epsilon$. Suppose we are in a node n where the minimizer is about to move. Then n is proved if and only if all its children are proved. In contrast, n is disproved if and only if at least one of its children is disproved. Suppose we are in a node n where the maximizer is about to move. Then n is proved if and only if at least one of its children is proved and n is disproved if and only if all of its children are disproved.

In the following we will consider n as being either a minimum ($\min$) or a maximum ($\max$) node. Let v be the value of n after n has been (dis)proved. Furthermore, let $\mathbf{s}$ be the state in n.

Case 1: n is a $\min$ node and n is proved

Hence, $b < h^*(\mathbf{s})$ and no pruning has been applied because all the children have to be proved for n to be proved. Then, we have $b < v \leq h^*(\mathbf{s})$. The first inequality is proved by contradiction. Assume $v \leq b$, then we have analogously to Case 3 in the proof of Lemma 4.10 that $v \geq h^*(\mathbf{s})$. Note that, despite the α-prune in Case 3 in Lemma 4.10, this particularly holds if none of the children of n are pruned. Together with the assumption $h^*(\mathbf{s}) > b$ this yields a contradiction to the assertion. Thus,

we have $b < v$. Analogously to Case 3 in Lemma 4.9 it follows that $v \leq h^*(\mathbf{s})$.

Case 2: n is a $\max$ node and n is disproved

Hence, $b \geq h^*(\mathbf{s})$ and no pruning has been applied. Then, we have $b \geq v \geq h^*(\mathbf{s})$. The first inequality is proved by contradiction. Assume $b < v$, then we have analogously to Case 4 in Lemma 4.9 that $v \leq h^*(\mathbf{s})$. Note that, despite the β-prune in Case 4 in Lemma 4.9, this particularly holds if none of the children of n are pruned. Together with the assumption $h^*(\mathbf{s}) \leq b$ this yields a contradiction to the assertion. Thus, we have $b \geq v$ and analog to Case 4 in Lemma 4.10 it follows that $v \geq h^*(n)$.

Case 3: n is a $\min$ node and n is disproved

Hence, $b \geq h^*(\mathbf{s})$ and at least one child was visited. Then we have $b \geq v \geq h^*(\mathbf{s})$. Let p be a disproved child (there is at least one) of n and $\mathbf{s}'$ be its state. Then p is a $\max$ node and we have $h^*(\mathbf{s}') \leq b - c(\mathbf{s}, \mathbf{s}')$ since p was disproved. Let v' be the value of p after it has been disproved. Then, the induction hypothesis (cf. Case 2) can be applied since $b - c(\mathbf{s}, \mathbf{s}') \leq b - \epsilon \leq b_0$ and therefore

$$\begin{aligned} h^*(\mathbf{s}') &\leq v' \leq b - c(\mathbf{s}, \mathbf{s}') \\ h^*(\mathbf{s}) \leq \quad h^*(\mathbf{s}') + c(\mathbf{s}, \mathbf{s}') &\leq v' + c(\mathbf{s}, \mathbf{s}') = v \quad \leq b. \end{aligned}$$

Case 4: n is a $\max$ node and n is proved

Hence, $b < h^*(\mathbf{s})$ and at least one child was visited. Then we have $b < v \leq h^*(\mathbf{s})$. Let p be a proved child (there is at least one) of n and $\mathbf{s}'$ be its state. Then p is a $\min$ node and we have $b - c(\mathbf{s}, \mathbf{s}') < h^*(\mathbf{s}')$ since p was proved. Let v' be the value of p after it has been proved. Then, the induction hypothesis (cf. Case 1) can be applied since $b - c(\mathbf{s}, \mathbf{s}') \leq b - \epsilon \leq b_0$ and therefore

$$\begin{aligned} b - c(\mathbf{s}, \mathbf{s}') &< v' \leq h^*(\mathbf{s}') \\ b < \quad v + c(\mathbf{s}, \mathbf{s}') = v &\leq h^*(\mathbf{s}') + c(\mathbf{s}, \mathbf{s}') \quad \leq h^*(\mathbf{s}). \end{aligned}$$

Therefore, the above implications for IPN iterations have been shown. The rest of the proof, i.e. termination and admissibility, is analog to the proof of Theorem 4.8. Note that as a consequence we keep proving the root until the second last iteration. Within the last iteration the root is disproved. □

Note, that this algorithm can easily be turned into a consistent algorithm by using a consistent heuristic h, pruning at $d = 0$ and caching with regards to d. Then $\text{ipn}(\mathbf{s}) = h^*(\mathbf{s}, d)$ (without any further assumptions).

There are several improvements possible to this generic version of IPN. First, the procedure that updates the proof numbers (see Figure 4.7) can be stopped whenever the proof and disproof numbers of a parent do not change. Submitting this parent to the subsequent call to select_most_proving_node, as suggested for pn-search [2], is not a good idea. When restarting the search from the root node, it might be possible to prune before reaching the suggested parent. Second, subtrees under (dis)proved

nodes can be deleted to reduce the space needs. Third, we can keep references to all identical states in a transposition table. When a node is (dis)proved, this transposition table can be used, to evoke updates on the transpositions and their parents.

The main drawback of IPN is its space complexity because the computed game tree has to be kept in memory. Even with the above enhancements the algorithm does a rather poor job on the cops and robber domain. This is because the game tree grows exponentially although the size of the move graph is well bounded. To face this problem, our implementation uses a version of depth-first proof-number search (df-pn-search) [60] rather than pure pn-search. Df-pn-search keeps a transposition table with stored (dis)proof numbers and searches similar to recursive best-first search. In fact, the order in which df-pn-search expands new nodes is the same as for pn-search. Unfortunately, df-pn-search is not complete for cyclic graphs and the move graph of the cops and robber domain has many cycles. This can be resolved by either using the general solution to the GHI [42] problem or by storing the states with respect to the g-cost at which they are encountered. The latter is more practical for the cops and robber domain since most states are only reencountered at a very small number of different depths. The only difference of our implementation to the original version of df-pn-search is that our algorithm always returns to the root node before traversing the transposition table to find the next node to be expanded. In contrast, df-pn-search recursively keeps exploring without necessarily returning to the root.

4.4 Reverse Minimax A*

Two-player game-tree search usually begins from a starting position and works towards possible goal positions, i.e. top-down. This search is difficult because the computation begins with one initial position, the root of the game tree, and computes towards multiple possible terminal nodes. From a single agent search perspective, the other way around, i.e. bottom-up, seems to be easier since computation works from multiple terminal states towards only one goal state. The underlying hope is better use of heuristics since only one goal is involved. Furthermore, bottom-up approaches have been used successfully in the computation of endgame databases, e.g. in checkers [71]. We develop Reverse Minimax A* (RMA*) which is a bottom-up algorithm and makes use of a backward heuristic. A retrograde analysis algorithm has been developed by Hahn and MacGillivray [33] (see Section 3.1 for an outline of an improved version). We use this algorithm to search for one particular given initial state and incorporate a heuristic function to speed up the search.

An outline of the algorithm in pseudo code is given in Figures 4.9 and 4.10. Consider Figure 4.11 as a sample computation. RMA* sets the value of the terminal states and pushes all their neighbors on an open queue (Figure 4.10(a)). Of course, this assumes that there is a finite set of terminal states and that they can be enumerated. This holds true in the cops and robber domain. In fact the number of terminal states in cops and robbers is bounded by kn^k since all the k cops can choose arbitrary positions in the graph and the robber has to be at the same vertex as one of the cops. (Using the

```
      function rma( g )
1       if g is a terminal state return terminalcost(g);
2       set_terminal_states_values(g);
3       while openqueue not empty
4         pop node p from openqueue that has smallest fcost;
5         s ← state(p);
6         if s = g return gcost(p);
7         if (cost_cache(s) not set) or
7a          (cost_cache(s) > gcost(p))
8           cost_cache(s) ← gcost(p);

9           if Min's turn in s
10            for all predecessors s' of s
11              v ← compute_max_value( s' );
12              if (cost_cache(s') is not set and v ≠ ∞) or
12a               (cost_cache(s') is set and cost_cache(s') > v)
13              push a node on openqueue with state ← s', gcost ← v, fcost ← v + h(g, s');

14          else // Max's turn in s
15            for all predecessors s' of s
16              v ← c(s', s) + gcost(p);
17              if (cost_cache(s') is not set) or
17a               (cost_cache(s') > v)
18              push a node on openqueue with state ← s', gcost ← v, fcost ← v + h(g, s');

19      return ∞;
```

Figure 4.9: Reverse Minimax A*.

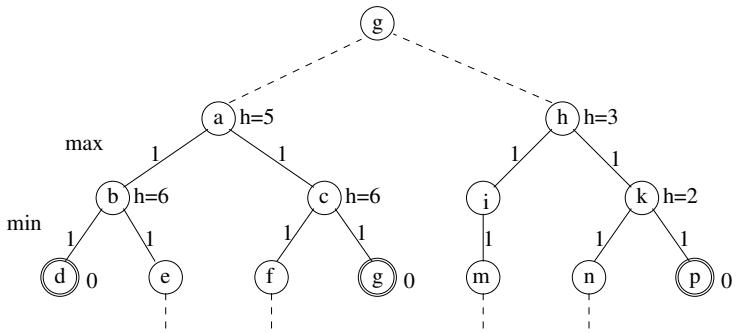
```
    procedure set_terminal_states_values( g )
1     for all terminal states s ∈ S
2       cost_cache(s) ← terminalcost(s);
3       if Max's turn in s // Min moved last
4         for all predecessors s' of s
5           push a node on openqueue with state ← s',
6           gcost ← c(s', s), fcost ← gcost + h(g, s');
7     return;
```

(a) Setting the values of all terminal states and pushing their neighbors onto the open queue.

```
    function compute_max_value( s )
1     r ← −∞; // result variable
2     for all successors s' of s
3       if cost_cache(s') is not set return ∞;
4       r ← max(r, c(s, s') + cost_cache(s'));
5     return r;
```

(b) Computation of the value of a max node in RMA*.

Figure 4.10: RMA* subroutines.



	cost_cache	openqueue
set_terminal_states_values	$(d,0),(g,0),(p,0)$	$(k,3),(b,7),(c,7)$
Expand k. Do not push h on queue since i is not set	$(d,0),(g,0),(p,0),(k,1)$	$(b,7),(c,7)$
Expand b. Do not push a on queue since c is not set	$(d,0),(g,0),(p,0),(k,1),(b,1)$	$(c,7)$
Expand c.	$(d,0),(g,0),(p,0),(k,1),(b,1)$	$(a,7)$

Figure 4.11: Part of the search tree and RMA*'s bottom-up computation.

techniques in Section 3.2 this bound can be reduced to $n\binom{n+k-2}{k-1}$). Note that this bound increases exponentially with the number of cops and hence computation becomes infeasible when many cops are used. Note further that when all agents are allowed to pass their turns, it is always the cops that move last in an optimal solution. Therefore, in set_terminal_states_values (Figure 4.10(a)) it is only necessary to push states onto the queue in which the minimizer has to move. When the agents cannot pass their turns, i.e. have to move to a different location in each step, all states have to be pushed onto the queue.

The open queue is sorted in increasing order of estimated path-cost, i.e. f-cost. Encountered states in which the minimizer is to move can be pushed onto the open queue immediately. Encountered states in which the maximizer is to move are delayed until all their successors are set (see Figure 4.11). This is done by giving back a value of ∞ in compute_max_value if not all the children are set and ignoring such a state in Line 12 (Figure 4.9). When all successors of a max state are assigned a value, the last successor that is set will push the max state onto the open queue since compute_max_value will then return the maximum of the values of all successors.

Theorem 4.12

Let h be an admissible backward heuristic. Let further $c(\mathbf{s},\mathbf{s}') > 0$ for all $\mathbf{s},\mathbf{s}' \in S(\mathbf{s} \neq \mathbf{s}')$ and $1 < |S| < \infty$. Then $\text{rma}(\mathbf{s}) = h^*(\mathbf{s})$.

Proof. We first prove that RMA*'s assigned values, via cost_cache, are upper bounds on the values of the game starting in the respective state. Note that if the value of the game starting in a state $\mathbf{s}$ is infinite, $\mathbf{s}$ will not be assigned a value due to Line 12 (Figure 4.9). We prove that if $\mathbf{s}$ is assigned a value then this value is an upper bound on the game starting in $\mathbf{s}$. This is proved by induction

beginning at the terminal nodes and inductively working our way up in the game tree. Therefore,

$$\text{cost\_cache}(\mathbf{s}) = \text{terminalcost}(\mathbf{s}) \geq h^*(\mathbf{s})$$

forms our induction basis. Note that cost_cache(s) is only set once for terminal nodes. For non-terminals cost_cache(s) might be set multiple times. However, as our induction basis we can assume that whenever cost_cache(s) is set for a state s (for the first time and thereafter), we have cost_cache(s) $\geq h^*(\mathbf{s})$.

Now let s be a state in which the minimizer is about to move. Let z be the $\max$ state from which s has been pushed onto the open queue. Note that s can be pushed onto the open queue multiple times but is only assigned a value, when s is first popped from the queue or a smaller g-cost is encountered. Without loss of generality we say z is the state that pushed the node with state s onto the queue that will be used to assign a value to s (at any given time). Then,

$$\begin{aligned} h^*(\mathbf{s}) &= \min_{\text{successors } \mathbf{s}' \text{ of } \mathbf{s}} (c(\mathbf{s}, \mathbf{s}') + h^*(\mathbf{s}')) \\ &\leq c(\mathbf{s}, \mathbf{z}) + h^*(\mathbf{z}) \\ &\leq c(\mathbf{s}, \mathbf{z}) + \text{cost\_cache}(\mathbf{z}) \\ &= \text{cost\_cache}(\mathbf{s}). \end{aligned}$$

Let s be a state in which the maximizer is about to move. Note first that s is only assigned a value if all its successors are set (to a value less than ∞). It follows that

$$\begin{aligned} h^*(\mathbf{s}) &= \max_{\text{successors } \mathbf{s}' \text{ of } \mathbf{s}} (c(\mathbf{s}, \mathbf{s}') + h^*(\mathbf{s}')) \\ &\leq \max_{\text{successors } \mathbf{s}' \text{ of } \mathbf{s}} (c(\mathbf{s}, \mathbf{s}') + \text{cost\_cache}(\mathbf{s}')) \\ &= \text{cost\_cache}(\mathbf{s}). \end{aligned}$$

We now prove that RMA* terminates. This is due to the fact that every state is only assigned a value finitely many times and is disregarded if no assignment takes place (Line 7 and 7a Figure 4.9). Let $\mathbf{s} \in S$. Whenever s is assigned a value if follows from the above that this value is an upper bound on the value of the game starting in s. Let $\epsilon = \max_{\mathbf{s}', \mathbf{s}'' \in S} c(\mathbf{s}', \mathbf{s}'') > 0$ (note that the maximum exists since $|S| < \infty$). Let M be the value that was first assigned to s. Then, at the moment of first assignment, the path computed from s to the terminal nodes is at most of length $d = \frac{M}{\epsilon}$. Therefore, there are at most

$$1 + |S| + |S|^2 + \ldots + |S|^d = \frac{|S|^{d+1} - 1}{|S| - 1} < \infty$$

paths that have a cost of less or equal to M. Furthermore, s will only be assigned a new value if the new encountered g-cost is smaller than the current. Hence, there are only finitely many values that can be assigned to s. This proves the termination of RMA*.

RMA* is complete, i.e. it assigns a value to a state s if and only if there is a solution for this state. Given an assignment of s by RMA* we know that this assignment is an upper bound, hence

a solution exists. If there is a solution, RMA* will assign a value since it eventually explores the entire game tree (regardless of the heuristic function) and terminates due to the above.
Now, we want to prove admissibility of RMA* when using an admissible backward heuristic h. If there is no solution for $\mathbf{s}$, i.e. the solution length is ∞, RMA* returns ∞ at the end of its iteration. Therefore, RMA* is clearly admissible in this case. In the case where there exists a solution, we will prove that the returned value is also a lower bound on the state's real value. RMA* terminates when the root $\mathbf{s}$ is first popped from the open queue. Let r be the node that has been popped. At that moment, we have

$$\text{fcost}(r) = \text{gcost}(r) + h(\mathbf{s}, \mathbf{s}) \leq \text{gcost}(p) + h(\mathbf{s}, \text{state}(p))$$

for all p on the open queue. Since $0 \leq h(\mathbf{s}, \mathbf{s}) \leq h^*(\mathbf{s}, \mathbf{s}) = 0$ we have

$$\text{gcost}(r) \leq \text{gcost}(p) + h(\mathbf{s}, \text{state}(p)) \leq \text{gcost}(p) + h^*(\mathbf{s}, \text{state}(p)). \tag{4.1}$$

The last term in (4.1) is a lower bound on the cost of a playout that terminates in a terminal state and goes through state(p). Hence, all possible playouts, induced by the remaining nodes on the open queue, that have not already been considered achieve a higher cost. Therefore, rma($\mathbf{s}$) $\leq h^*(\mathbf{s})$.
In summary, if $h^*(\mathbf{s}) = \infty$ then rma($\mathbf{s}$) $= \infty$, if $h^*(\mathbf{s}) < \infty$ then rma($\mathbf{s}$) $\leq h^*(\mathbf{s}) \leq$ rma($\mathbf{s}$). □

When searching in a single-agent domain with an algorithm like A*, an admissible and consistent heuristic will never re-expand nodes during the search. We encountered a similar phenomena with RMA*. When using the heuristic from Section 4.5, RMA* did not have to re-expand nodes. Unfortunately, this heuristic is not consistent in the single-agent search manner, i.e. the heuristic can differ by more than the edge cost (in the move graph) from one state to the next. The immediate assumption that RMA* never re-expands nodes is wrong. When using an admissible heuristic and randomly introducing zeros for some states, RMA* had to re-expand nodes. Correct computation and the re-expansion is ensured by Lines 7a, 12a and 17a (Figure 4.9). More research is required to fully understand the properties of a backward heuristic that are needed to keep RMA* from re-expanding nodes.

4.5 Experiments

We test the algorithms in the previous sections on 1-cop-win graphs and 2-cop-win graphs. We measure the performance of the algorithms in terms of node expansions, nodes touched and computation time[1].

We set all edge costs to one and the cost for passing a turn is set to one as well. All the algorithms require forward or backward heuristics. Recall that a forward heuristic is an estimate on the number of turns that the cops and robber take until capture. An estimate can be computed by assuming the

[1] All experiments were run on a Intel Xeon© 2.5GHz CPU with 16GB RAM.

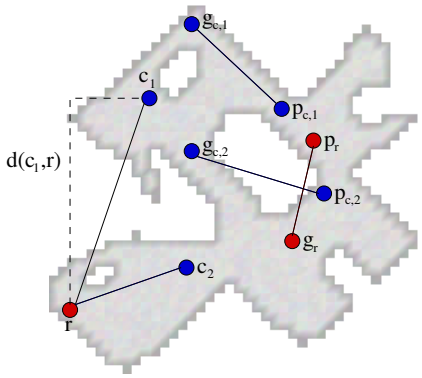


Figure 4.12: Visualization of the different distance measures used to compute admissible forward and backward heuristics.

robber stays still and the cops move towards him. The distance between the cops and robber can be estimated using a distance metric, what would normally be a heuristic for single-agent search (for a visualization see Figure 4.12, the robber is at r and the cop at c_1 and $d(c_1, r)$ measures the distance of these two positions). Let d be such a distance metric that is consistent in the single-agent search sense, i.e. $d : V(G) \times V(G) \to \mathbb{R}$ and $d(v, w) \leq d(v', w) + c(v, v')$ for all $vv' \in E(G)$. Then, the following forward heuristic is consistent and admissible.

$$h(\mathbf{s}) = \begin{cases} 0 & \mathbf{s} \text{ is a terminal state} \\ 2 \max_{1 \leq i \leq k} d(\mathbf{r}, \mathbf{c}_i) - \text{turn}(\mathbf{s}) & \text{otherwise, where } \mathbf{s} = (\mathbf{r}, \mathbf{c}_1, \ldots, \mathbf{c}_k, \text{turn}(\mathbf{s})) \end{cases}.$$

The factor of two is because the cops only get to move once every two turns while the robber stays still. Therefore, a distance metric underestimates by approximately a factor of two, depending on whose turn it is.

For RMA* we used the following admissible backward heuristic: Let $\mathbf{g}_r$ and $\mathbf{g}_{c,i}$ be the goal (i.e. initial position of the game play), $\mathbf{p}_r$ and $\mathbf{p}_{c,i}$ the current positions of the robber and the cops ($1 \leq i \leq k$) as indicated in Figure 4.12. We assume all agents traveled from their initial position towards their current position on a shortest path. The shortest path itself can be estimated with a given distance metric d. Hence, let d_r and $d_{c,i}$ be the distance from the robber and the cops to their goal locations, i.e. $d_r = d(\mathbf{p}_r, \mathbf{g}_r)$ and $d_{c,i} = d(\mathbf{p}_{c,i}, \mathbf{g}_{c,i})$. Then,

$$h((\mathbf{p}_r, \mathbf{p}_c, t), (\mathbf{g}_r, \mathbf{g}_c, g_t)) = \max_{1 \leq i \leq k} \begin{cases} 2 \max\{d_r, d_{c,i}\} & t = g_t \\ 2d_r + 1 & d_r = d_{c,i}, t \neq g_t \\ 2d_{c,i} - 1 & d_r < d_{c,i}, t = 1, g_t = 0 \\ 2d_{c,i} + 1 & d_r < d_{c,i}, t = 0, g_t = 1 \\ 2d_r + 1 & d_r > d_{c,i}, t = 1, g_t = 0 \\ 2d_{c,i} - 1 & d_r > d_{c,i}, t = 0, g_t = 1 \end{cases}. \tag{4.2}$$

is an admissible backward heuristic.

When using this heuristic with the optimizations in Section 3.2, i.e. when the positions of the cops are not distinguished, it is not possible to compute $d_{c,i}$ correctly since the ith cop in the current position is not necessarily the ith cop in the goal position. Hence, the lower bound

$$d_{c,i} = \min_{1 \leq j \leq k} d(\mathbf{p}_{c,i}, \mathbf{g}_{c,j}) \tag{4.3}$$

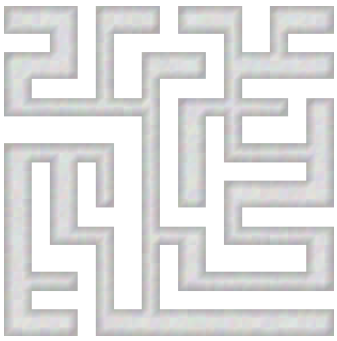


Figure 4.13: A 40x40 map from the third set.

is used together with (4.2).

When only one cop is considered ($k = 1$), the above heuristics differ much in accuracy depending on the choice of the distance metric d. This is due to the fact that there are no obstacles present and hence the maps have a somewhat maze-like structure (see Figure 4.13). The more accurate the metric d, the better the quality of the heuristic. Therefore, we experiment with three choices for d. First, we consider the uninformed heuristic, i.e. $d \equiv 0$. Second, let (v_x, v_y) and (w_x, w_y) be the coordinates of v and w, respectively. Then, the maximum norm distance metric is defined as

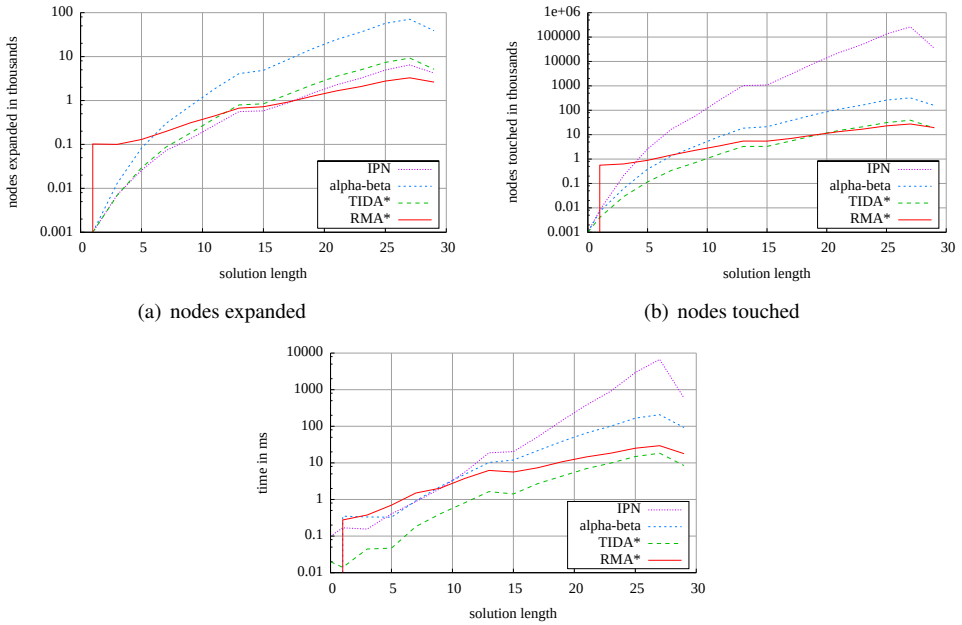
$$d_\infty((v_x, v_y), (w_x, w_y)) = \max\{|v_x - w_x|, |v_y - w_y|\}.$$

Third, we consider the perfect distance metric d_{perfect} that returns the exact graph distance of two vertices. When using the uninformed metric in RMA*, we refer to the algorithm as RA. It is referred to as normal or improved RMA* when d_∞ or d_{perfect} is used, respectively. Equivalently, TIDA* is referred to as normal or improved when d_∞ or d_{perfect} is used, respectively. The perfect distance metric can be expensive to compute. But, as will be seen, the experimental results suggest that a good heuristic can have a large effect on the work required to compute optimal solutions.

When multiple cops are used, the minimization in (4.3) effectively nullifies the improvements of a better distance metric. Moreover, due to the structure of the maps, usual approximate distance metrics, e.g. the maximum norm distance metric, are more accurate in this case. Hence, we only report the results for the above distance metrics for verification although analysis with d_∞ would be sufficient.

4.5.1 1-cop-win graphs

We tested maps of size 15x15 (first set), 20x20 (second set), 40x40 (third set) and 60x60 (fourth set). One representative map from the third set is plotted in Figure 4.13. Set one and two contain 15 maps each. Set three and four contain 20 maps each. On each map, we generated 1000 random initial locations for the cop and the robber. To guarantee termination of the experiments with IPN we restricted the solution length to a maximum of 31 while generating the problem instances for set one. Since in a computer game the cop and the robber can be spawned anywhere in the map, our



(a) nodes expanded

(b) nodes touched

(c) computation time

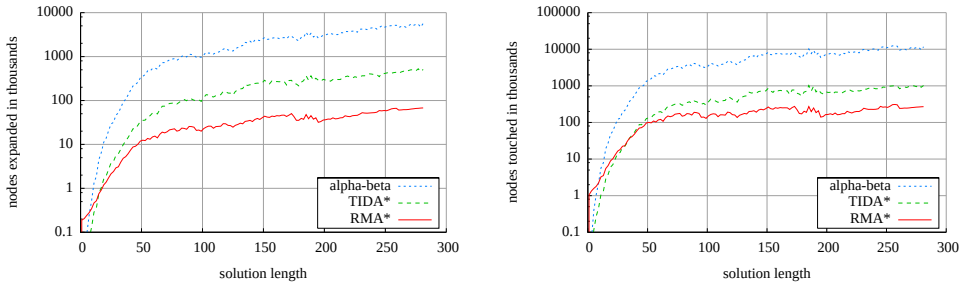
Figure 4.14: Performance of IPN, α-β , TIDA* and RMA* on set one (15×15 maps).

focus is on solving the game for any initial position. Therefore we sampled the initial locations at random. If we focused on the mathematical definition of the cops and robber problem, the initial location selection would have to be included into the robber's strategy.

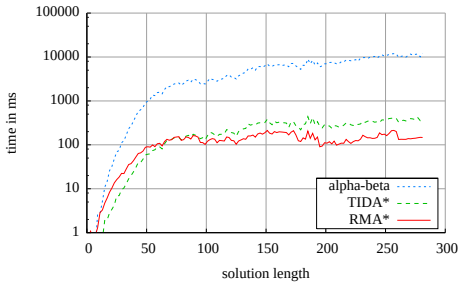
We report the number of nodes expanded, nodes touched and computation time plotted against the solution length. The solution length can also be interpreted as the depth of the search tree that has to be searched to find a solution. To generate meaningful statistics, all values for set three and four are grouped together into buckets of 5 and are averaged over these buckets. For set one and two the values are directly averaged over the instances for each solution length. Plots for all sets can be found in Figures 4.14-4.17.

On the first set of maps we can observe that the number of node expansions in IPN is less than the number of node expansions in TIDA* (Figure 4.14(a)). This is because IPN only expands nodes that have a high chance of proving the root's value and it has actually been designed to expand as few nodes as possible. However, the difference is relatively small even though we plot using a logarithmic scale. Due to the update operations of the (dis)proof numbers in each iteration IPN clearly touches more nodes than any other algorithm (Figure 4.14(b)) since it iterates over the entire expanded game tree. This gets particularly significant when the depth increases and the game tree grows exponentially. Furthermore, this causes a high increase in computation time compared to all other algorithms and in particular to TIDA* (Figure 4.14(c)).

Since node expansions are computationally cheap in the cops and robber domain, IPN is not the best algorithm to use. Nonetheless, when coping with domains where node expansions are

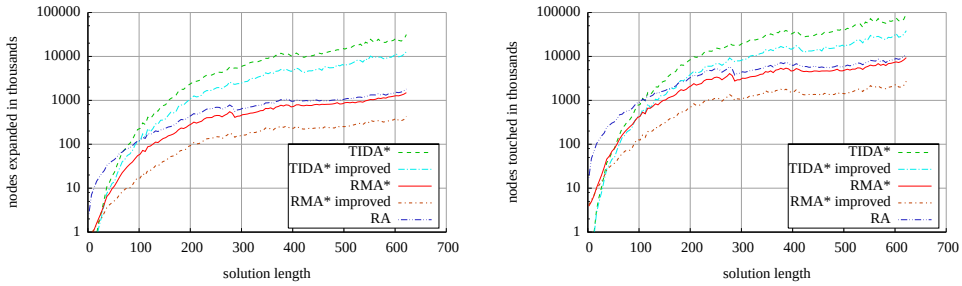


(a) nodes expanded (b) nodes touched

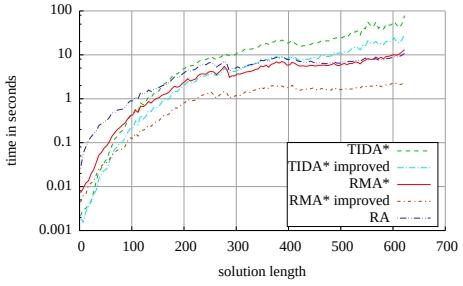


(c) computation time

Figure 4.15: Performance of α-β , TIDA* and RMA* on problem set two (20×20 maps).

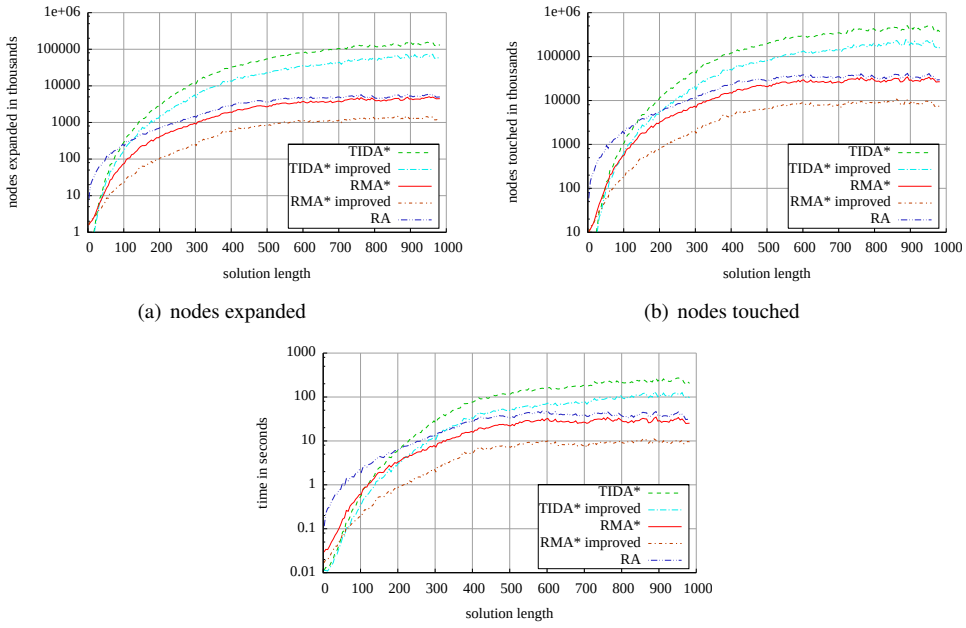


(a) nodes expanded (b) nodes touched



(c) computation time

Figure 4.16: Performance of TIDA*, TIDA* improved, RA, RMA* and improved RMA* on problem set three (40×40 maps).



(a) nodes expanded

(b) nodes touched

(c) computation time

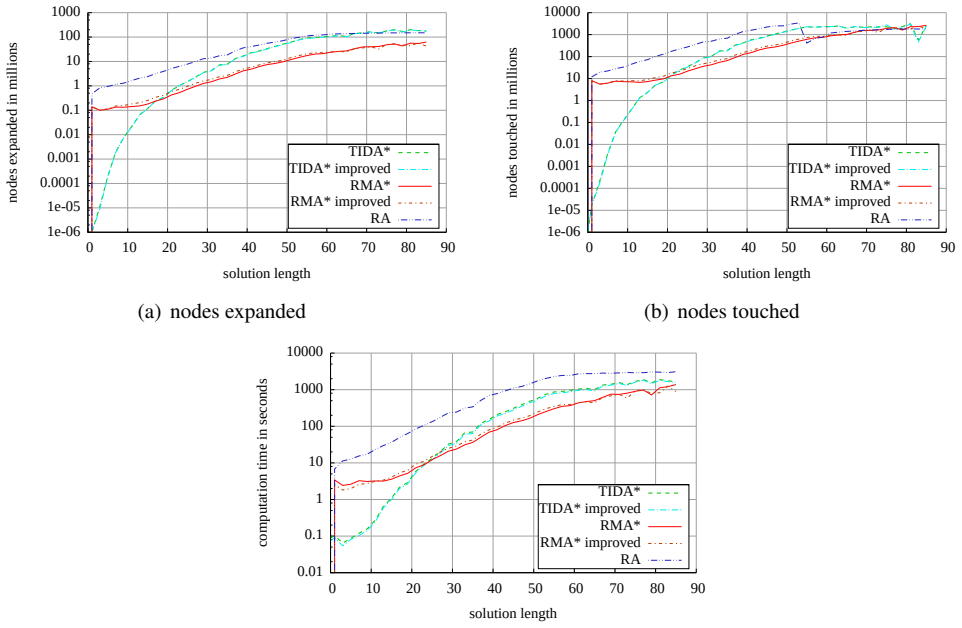
Figure 4.17: Performance of TIDA*, TIDA* improved, RA, RMA* and improved RMA* on problem set four (60×60 maps).

expensive and the state space is large (which means that there is a large number of goal states, rendering RMA* inapplicable), IPN might do much better since it is the algorithm out of the three top-down approaches that expands the fewest nodes.

When testing with the first set of maps, we can already perceive the increase of node expansions in the α-β algorithm compared to the other approaches. Although transposition tables are used, this increase still seems to be exponential in the depth of the computed game tree. It can be seen from Figure 4.15 that α-β expands and touches an order of magnitude more nodes than TIDA* and RMA*, causing much higher runtimes on the second set of maps.

Since we implemented RMA* with a standard priority queue, the cost of maintaining the queue in sorted order and having to push up all terminal states before starting the search dominates the actual search time. Therefore, for small solution lengths, TIDA* dominates RMA* (see Figure 4.16(c)). Within computer games, edge costs are often approximated to be able to bucket path costs within priority queues, e.g. for path finding. Under this assumption, the f-costs in RMA*'s priority queue can be sorted in buckets and access to the priority queue can be realized in constant time. It can be seen from Figure 4.16 that TIDA* expands and touches an order of magnitude more nodes than RMA* when used for problems with long solution lengths from the third set of maps. Here, even using an unoptimized open queue in RMA* pays off compared to TIDA* with respect to time (Figure 4.16(c)).

Our experiments on the third and fourth set (Figures 4.16 and 4.17) reveal another interesting fact. When used with maximum norm distance metric RMA* yields similar performance as with



(a) nodes expanded

(b) nodes touched

(c) computation time

Figure 4.18: Performance of TIDA*, TIDA* improved, RA, RMA* and improved RMA* on 2-cop-win graphs.

the uninformed metric. This is due to the nature of our test maps that are all scaled mazes. Here, d_∞ is highly inaccurate. Therefore, when used with perfect distance metric, RMA* scales much better yielding speedups by more than a factor of three. Since maps that inherit k-cop-win graphs can have obstacles and thus are not scaled mazes anymore, d_∞ will be far more accurate on these maps. Therefore, it is to be expected that normal RMA* will experience greater speedups compared to RA on such maps (see Subsection 4.5.2).

The largest maps we have solved here are of size 60x60. Solving larger maps is possible but prolongs statistically significant experiments considerably. It can be seen in Figure 4.17(c) that RMA*'s computation time requirement is very low when used with an adequate heuristic. RMA* improved only needs about 10s to compute solutions that involve more than 500 steps, i.e. the depth of the game tree is greater than 500.

4.5.2 2-cop-win graphs

We also like to evaluate the performance of the algorithms on more complex maps. Here, only RMA* and TIDA* will be evaluated since IPN and Minimax have already proved to be unsuitable within the previous experiments. We used four maps from the commercial game Baldur's Gate, the smallest having 175 and the largest having 558 vertices. When choosing octile maps (cf. Definition 2.8) as the underlying graph representation all these maps become 2-cop-win. We generated 1000 random initial positions for two cops and one robber for each map. Furthermore, we used the optimizations from Section 3.2.

Again, we measured the number of node expansions, nodes touched and computation time necessary to solve the problems. The results are averaged over the solution length and are depicted in Figure 4.18. TIDA* and TIDA* improved perform very similar and have the same curves in these plots. The trends remain the same as in the 1-cop-win graph experiments. However, we can perceive that using a better distance metric for the construction of the forward and backward heuristic does not gain better performance of the algorithms. This is, as explained above, due to the fact that d_∞ is more accurate on these more natural maps, hence the difference between d_∞ and d_{perfect} is small, and the minimization in (4.3) counteracts the improvement in the distance metric. Hence, the overhead of computing d_{perfect} is not justified when multiple cops are used. However, using heuristic guidance is highly important since the RA algorithm, i.e. RMA* with uninformed heuristic, performs significantly worse than RMA* with heuristic guidance due to d_∞ or d_{perfect}.

The above experiments show that our optimal approaches scale well with respect to the map size. However, analog to the RA algorithm, it is apparent that computation time requirements are, despite all improvements, impractical for online use in modern computer games particularly when multiple cops are used (cf. Figure 4.18(c)). Hence, there is a need to move away from optimal to suboptimal approaches which we will do in the next chapter.

4.6 Summary and open problems

Within this chapter we have explored algorithms to compute optimal solutions when one initial position is given. We investigated three top-down algorithms, Minimax, Two-Agent IDA* and Iterative Proof-Number Search. We showed how the later two algorithms can be enhanced with caching. Additionally, we used iterative bound increase in Iterative Proof-Number Search. These modifications enable feasible application of the algorithms to the cops and robber domain. Furthermore, we developed a new algorithm, Reverse Minimax A*, that introduces heuristic search into the retrograde analysis algorithm from the previous chapter.

Our comprehensive experiment on multiple types of maps showed that our algorithm, RMA*, outperforms all other approaches for long solutions lengths. For small solution lengths, TIDA* is the method of choice since RMA* performs an initial overhead computation to assign all terminal states and keeps a sorted priority queue.

It is an open question when RMA* does not have to re-expand nodes, i.e. what the properties of an admissible backward heuristic are that guarantees such behavior. Furthermore, we suspect that it is possible to decrease the initial overhead computation in RMA* of assigning a value to all terminal states and pushing their neighbors onto the priority queue before the search begins. Since not all terminal states might be reachable from the initial state via optimal play, it seems likely that only a small subset of terminals have to be considered initially and if needed more terminals can be set later during the search. More research is needed to solve these problems.

Chapter 5

Approximative algorithms

The mistakes are all waiting to be made.

Chess master Savielly Grigorievitch Tartakower (1887-1956) on the game's opening position

Today's computer games are played on very large maps but have tight bounds on resource usage, especially computation time. Although the algorithms used in Chapter 4 scale well with increasing map size, their computational requirements become infeasible on very large maps. Therefore, computing optimal policies is not practical for usage in modern computer games or any other application that requires solutions in realtime. This gives rise to two problems. First, policies have to be computed quickly. Second, the computed strategies should exhibit intelligent behavior. This can be achieved by computing near optimal strategies, hence our goal is to compute very good approximations of an optimal strategy. If less optimal strategies are needed, it is always possible to degrade a near optimal approach.

Within this chapter, we will outline previous approaches, including Cover [36], Dynamic Abstract Minimax (DAM) [15], a random beacon algorithm, minimax and hill climbing with distance heuristic. Then, we will introduce two new approaches, *TrailMax* and *Dynamic Abstract TrailMax* (DATrailMax), that also compute approximate solutions. To measure the performance of the algorithms various experiments will be conducted.

Besides the above mentioned approaches there are various learning and any-time algorithms in the literature [38, 39, 44]. However, these algorithms require repetitions of the capture scenario to learn how to capture a target or how to run away from a pursuer. Since in computer games the algorithms have to work on potentially unknown maps immediately, our focus is on algorithms that compute move policies without having to repeat the game. Hence, we do not study these learning algorithms here.

As this thesis includes the first study of optimal algorithms, previous work in cops and robbers/moving target search has not, whether for the pursuer or the target, compared its methods against optimal policies. This thesis is the first to conduct a study of all the above target algorithms with respect to their achieved suboptimality. Isaza *et al.* [37] compared their pursuer algorithm,

Cover with Risk and Abstraction (CRA), to a best-response against some of the here studied target algorithms. However, they do not study how exploitative these target algorithms are. Using the RA algorithm (cf. Section 3.1) we compute a best-response for the above target algorithms and conduct experiments to evaluate their exploitability.

Since our focus is on the target and the cop is potentially played by a human player, we concentrate on the one-cop-one-robber problem here. However, all the following methods can easily be extended to multiple cops. Furthermore, we are interested in playing on typical video game maps that include obstacles. Hence, one cop cannot catch a robber that plays optimal when both agents play with the same speed. To enable execution of experiments, i.e. many simulations of the game, we have to decide between one of the three ways to guarantee termination: the target moves suboptimally from time to time, the game is ended after a certain number of steps, or the cop is faster than the target. The first possibility contradicts our wish to compute near-optimal policies for the robber. The second choice is problematic due to the choice of timeout conditions. Furthermore, it does not measure the full amount of suboptimality generated by a given strategy because the game is truncated after the timer runs out. Moreover, it is easy to construct an algorithm that achieves optimal results in this game: detect all cycles around obstacles of length greater or equal to four in the map, run to a cycle where the cop cannot capture the robber before reaching the cycle, and exploit the cycle. Therefore, we allow the cop to be faster than the robber. For simplicity we allow the cop to make d subsequent moves when the robber only gets one, i.e. to move to any location within a radius of d of his current position.

In the following, we will subsequently discuss the random beacon algorithm in Section 5.1, Dynamic Abstract Minimax in Section 5.2, Cover and our variation in Section 5.3, and our two new methods TrailMax and Dynamic Abstract TrailMax in Section 5.4. After, we will conduct an experiment where all these target algorithms, plus Minimax and hill climbing with the distance heuristic, are run against a Nash equilibrium playing cop in Subsection 5.5.1. This measures the suboptimality of the algorithms and we study their computational performance during these experiments. To evaluate the algorithms' exploitability, we compute best responses for some of these algorithms in a second experiment in Subsection 5.5.2. We further discuss the differences in results between our experiments and other known results for the algorithm Cover in Subsection 5.5.3.

5.1 Random Beacon Algorithm

Heuristics are estimates on the value of the game starting in given initial positions. These estimates need to be computed very quickly. Hence, heuristic evaluation is fast and can be done several times without great impact on computation time. A fast greedy algorithm can therefore perform hill climbing with a given heuristic function. That means, given a current state, the algorithm evaluates all successor states with the given heuristic function and chooses to move to one of the, possibly multiple, best successors. This is the method underlying the Greedy and Cover algorithm within the

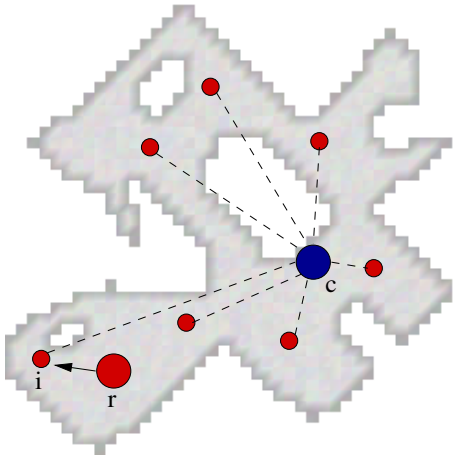


Figure 5.1: Illustration of the random beacon algorithm. r and c signify the positions of the robber and cop, respectively. The small dots are the positions of the beacons.

experiments in Section 5.5. Greedy uses the consistent forward heuristic from the previous chapter and Cover uses the Cover heuristic (cf. Section 5.3).

The random beacon algorithm is an improvement over a greedy method. It uses a number of beacons that it distributes over the map. Consider Figure 5.1 for an illustration of the computation. The big dot labeled with r is the current position of the robber and the big dot labeled with c represents the position of the cop. Beacons, the small dots, are distributed randomly across the map. Then, each beacon is evaluated using its heuristic distance to the cop's position, the dashed lines. Afterwards, the robber chooses to run to the beacon that is heuristically furthest away from the cop's position. He then plans a path towards this beacon with PRA* [76].

It is crucial to the algorithm that beacons are distributed uniformly over the map. For example, consider the situation where the beacon i (Figure 5.1) would not be sampled. In this case the robber will have to move towards the cop since all other beacon positions point in that direction. Furthermore, when the robber is in an outlier, e.g. the edge of the map or a corner, he will move into the inside of the map with high probability since most beacons will be sampled there. Besides other problems (cf. Subsection 5.5.1), this can cause very suboptimal behaviour particularily in the endgame.

The number of beacons that the algorithm distributes is a parameter that we will call b in the following. Furthermore, we introduce a second parameter. Since the result of a computation is a path that has been generated by PRA*, we choose to follow this path for a number of m steps. Hence the name *randombeacons(b, m)*. Note that it is also possible to choose a fraction of the path length that will be followed rather than an absolute number of steps. However, we do not investigate this alternative here.

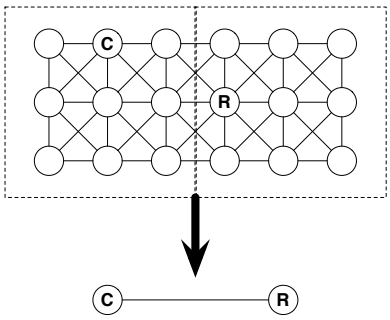


Figure 5.2: Map abstraction for DAM.

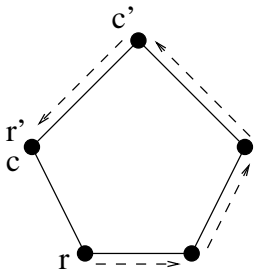


Figure 5.3: DAM computation on cycles.

5.2 Dynamic Abstract Minimax

Dynamic Abstract Minimax (DAM) was first proposed by Bulitko and Sturtevant [15]. This algorithm assumes that various resolutions of abstract maps are available, where an abstract map is created by taking sets of states in an original map and merging them together to form a more abstract map (see Definition 7.11 for the mathematical definition of an abstraction). DAM chooses an initial level of abstraction and then computes a minimax solution to a fixed depth. If the robber cannot avoid capture at that level of abstraction, computation proceeds to the next lower level of abstraction. We illustrate this in Figure 5.2. In the abstract map two sets of 9 states have been abstracted together to form a 2-node graph. The cop can catch the robber in one move in the abstract graph, so DAM will search again on the lower level of abstraction. Assume there are ℓ levels of abstraction and the cop and the robber occupy distinct nodes up until level m. The original algorithm begins planning at level m. Running the experiments in Section 5.5 for multiple fractions of m showed that starting at level $m/2$ is superior. We report the results for m, $\frac{3}{4}m$, $\frac{1}{2}m$ and $\frac{1}{4}m$.

If the robber can escape, an abstract goal destination is selected and projected onto the actual map. PRA* is used to compute a path to that node which is subsequently followed for one step. Since only the goal destination is projected onto the ground level, DAM can make mistakes when cycles exist in the strategy. For example, consider the cycle depicted in Figure 5.3. Let the robber and cop be on the positions indicated with r and c, respectively. The solution is to run around the cycle, indicated by dashed lines, but after four steps for the robber and four steps for the cop,

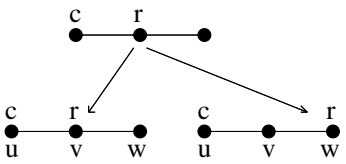


Figure 5.4: Example where the original tie breaking of the cover heuristic computation can cause the robber to remain in v instead of going to w.

the robber will reach the initial position of the cop. Hence, when computing with depth eight, the robber will run towards the cop. An immediate solution to this problem is to make DAM only refine one abstract step. However, running the experiments in Section 5.5 for such a variant showed that the original algorithm, despite its flaws, achieves slightly better results. Nonetheless, we report the results for both variants.

We use the same idea of using abstractions to speedup our own work. Our algorithm, DATrail-Max, selects level $m/2$ as the first level of abstraction, solves the problem on this level and proceeds to the next lower level if the robber cannot survive long enough. Otherwise, the abstract solution path is refined into a ground level path.

5.3 Cover

The Cover algorithm, as a state-of-the-art algorithm for moving target search, has been used for both, the cop and the robber [37]. It consists of a heuristic combined with a hill climbing algorithm. The heuristic evaluates a given state, i.e. the positions of cops and robber, for an agent. This is done by computing the area of the graph that the agent can reach before any other agent. The heuristic returns the number of nodes of this area. Hill climbing then evaluates the heuristic for each possible successive position of the agent and chooses the position with the maximal heuristic value. The algorithm is motivated by the proof that three cops are sufficient to catch a robber in a planar graph [1] where covered area plays an important role.

The original algorithm breaks ties by assigning the nodes on the border between two covered areas to the cop. However, this causes the heuristic to be inaccurate for the cop even for simple problems like a 1-dimensional path of length 4 or 5. When using the heuristic for the pursuers, Isaza *et al.*[37] use a notion of risk to increase the pursuer's aggressiveness and circumvent this inaccuracy for the cop. Nevertheless, such enhancements were not used with Cover as a target algorithm.

As an example of inaccuracy for the robber, consider the graph in Figure 5.4. There are three vertices, u, v, and w. The cop starts on u, the robber on v, and it is the robber's turn. When the robber remains on v, v and w are considered robber cover. If he moves to w, u and v are cop cover (due to the tie-breaking rule) and only w is robber cover. Thus, when maximizing, the robber prefers to stay in v, which is suboptimal.

In this work, we have used two alternate versions of Cover that eliminate this problem. The first

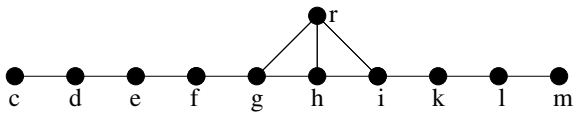


Figure 5.5: Graphs for explaining our alternate versions of Cover.

version generally does the same computation as Cover, i.e. is implemented by running simultaneous instances of Dijkstra's algorithm around the agents' positions until they interfere with the regions declared as the opponent's cover. Only, the tie breaking at the borders is modified such that vertices are only declared robber cover if the robber is guaranteed to reach them no matter what the cop does. As an example, reconsider the graph in Figure 5.4. We first evaluate the possible move of the robber to go to w. No matter what the cop does in his next move, the robber can still reach v. This would effectively mean that the robber runs into the cop, which is not optimal but this possibility is essential for the computation. Thus, v and w are declared as robber cover. Now, assume the robber stayed on v. Then, the cop can catch the robber in his next turn by moving to v. Hence, the robber will not be able to reach w no matter what the cop does and only v is declared as robber cover. Therefore, the above problem is resolved because the robber will choose to move to w instead of remaining at v. This version proved to be slightly better than the original Cover algorithm (in the sense of the experimental evaluation in Subsection 5.5.1) but we do not report its results here. However, a subset of the experimental setup in Subsection 5.5.1 has been used in [57] to evaluate this alternative computation.

In the above described first variation, the rule that only vertices are declared robber cover if he is guaranteed to reach them no matter what the cop does, is only applied to the tie breaking in between covered areas. When the cops and the robber move at different speeds further improvement can be made by applying this rule to the entire computation. This is our second variation, whose results we report in this thesis. To explain the difference between these two approaches, consider the graph in Figure 5.5. Let the cop be at c and the robber evaluates his move to r (coming from some other parts of the graph not depicted here). Hence, it is the cop's turn next. Furthermore, let the cop move at twice the speed of the robber. Then, the computation proceeds as follows:

	original	first variation	second variation
1	cop expands c and declares it as cop cover		
2	robber expands r and declares it as robber cover		
3	cop expands d and e and declares them as cop cover		
4	robber expands g, h and i and declares them as robber cover		
5	cop expands f and declares it as cop cover		
	cop does not expand g since already covered by the robber	cop expands g	
6	robber expands k and declares it robber cover		
7		cop does not expand r, h or i since they are in the interior of the robber cover	cop expands r, h and i
8	robber expands l and declares it robber cover		
9			cop expands k and l
10	robber expands m and declares it robber cover		robber does not expand m since the parent l has already been caught by the cop
11			cop expands m and declares it as cop cover

Hence, the cop cover in the original and our first variation is $\{c, d, e, f\}$ whereas it is $\{c, d, e, f, m\}$ in the second variation. The latter is better since m cannot be reached by the robber when the cop runs towards m. However, as can be seen in Step 7, the algorithm is required to continue to expand vertices for the cop even if they are in the interior of the covered area by the robber. This is computationally more expensive.

The new algorithm is depicted in pseudo code in Figure 5.6. Note, that we have improved a naive implementation by stopping the computation whenever the robber has no remaining new nodes scheduled for expansion (Line 6, Figure 5.6). Since the cop never stops expanding new nodes until the entire graph is explored it is always the robber who first runs out of nodes to expand. Due to this stopping condition and despite the above computation overhead of having to expand nodes for the cop even in the interior of the covered area by the robber, this new version expands fewer nodes than the original Cover algorithm in the experiments in Section 5.5. We only return the robber cover in Line 31 (Figure 5.6). If the computation is needed for the cop this can be easily adjusted by returning the difference of the robber cover and the number of vertices in the graph. Note that returning cop_cover does not yield a complete computation since we stop map exploration prematurely and the cop might not yet have explored all the vertices that he covers.

When being used for the pursuer, Cover with Risk and Abstraction (CRA) [37] makes use of abstractions to decrease computation time and to scale to large maps. This has not been used for the robber. Due to the nature of the algorithm it is clear that using abstraction is only an enhancement to gain speed. The heuristic is most accurate with full information about the problem, thus when used without abstraction. Within our experiments (cf. Section 5.5), the Cover heuristic without abstractions did not perform as well as other approaches in terms of survival time against a Nash equilibrium playing cop. Therefore, we did not extend the algorithm to incorporate abstractions.

```
    function cover( r, c, turn )
     // r is the robber's position, c is the cop's position, turn specifies who moves next
1      if( r = c ) return 0;
2      push node on cop_queue with v ← c, gcost ← 0;
3      push node on robber_queue with v ← r, gcost ← 0;
4      robber_cover ← 0;
5      cop_cover ← 0;
6      while( robber_queue not empty )
7        qrobber ← node with best gcost on robber_queue;
8        qcop ← node with best gcost on cop_queue;
9        if( (turn = 1 and qrobber.gcost < qcop.gcost ) or
10           (turn = 0 and qrobber.gcost ≤ qcop.gcost ) )
11         // robber moves
12         take qrobber from robber_queue;
13         if( (qrobber.v not in robber_closed) and (qrobber.parent not in cop_closed) )
14           insert qrobber.v into robber_closed;
15           if( qrobber.v not in cop_closed ) // the robber moved into the cop and stole a covered vertex
16             cop_cover ← cop_cover - 1;
17           robber_cover ← robber_cover + 1;
18           for all w ∈ N(qrobber.v)
19             if( w not in robber_closed )
20               push node on robber_queue with v ← w, gcost ← qrobber.gcost + c(qrobber.v, w), parent ← qrobber.v;
21       else
22         // cop moves
23         take qcop from cop_queue;
24         if( qcop.v not in cop_closed )
25           insert qcop.v into cop_closed;
26           if( qcop.v not in robber_closed )
27             cop_cover ← cop_cover + 1;
28           for all w ∈ N(qcop.v)
29             if( w not in cop_closed )
30               push node on cop_queue with v ← w, gcost ← qcop.gcost + c(qcop.v, w);

31     return robber_cover;
```

Figure 5.6: Pseudo code for our redefinition of the Cover heuristic.

5.4 TrailMax and Dynamic Abstract TrailMax

We now outline our approach to computing near-optimal move policies for the robber. We will first motivate the algorithm and then provide more details. For ease of understanding the following ideas will be developed for the game where the cop and robber move with the same speed. However, all the definitions and theorems are extensible to different speed games.

The robber makes the assumption that the cop knows where he is going to move, i.e. that the cop will play a best response against him. Under this assumption, the robber tries to maximize the time to capture. This can also be interpreted as "running away", i.e. taking the path that the cop takes longest to intersect. We will now formalize this idea. A path is a trajectory an agent can take when moving through the graph. Therefore, let

$$P(v) = \{p : \mathbb{N} \to V(G) | p(0) = v, \forall i \geq 0 : \ p(i+1) \in N[p(i)]\}$$

be the set of paths through graph G starting in v. Given paths p_r and p_c that the robber and cop follow disregarding the opponent's actions, we can compute the number of turns both agents take until capture occurs:

$$T(p_r, p_c) = \min(\{2t | t \geq 0, p_c(t) = p_r(t)\} \cup \{2t-1 | t \geq 1, p_c(t) = p_r(t-1)\}).$$

Definition 5.1 (TrailMax)
Let $v_r \in G$ and $v_c \in G$ be the positions of robber and cop in G. We define

$$\text{TrailMax}(v_r, v_c) = \max_{p_r \in P(v_r)} \min_{p_c \in P(v_c)} T(p_r, p_c). \tag{5.1}$$

Recall that a graph G is called k-cop-win if k cops have a winning strategy on G for any initial position of the cops and the robber and when all agents move with same speed.

Theorem 5.2
Let G be a 1-cop-win octile map. Let v_r and v_c be the initial positions of the robber and cop. Then TrailMax(v_r, v_c) returns the optimal value of the game where the cop and robber move at the same speed.

Proof. 1-cop-win graphs are well classified by the subsequent removal of pitfalls. Hence, it seems possible to prove the theorem by induction on the number of vertices in the graph. Since we are dealing with octile maps, there are only two cases that have to be considered. These two cases are depicted in Figure 5.7 where the circle symbolizes the current graph and n is the new added vertex. However, this results in an extensive definition of various subcases that we omit here. In contrast, we give a more compact alternative proof.
Let $d(v, w)$ denote the graph distance of a vertex v and w in G, i.e. the minimal number of edges that separate v from w. Let $B(v, d) = \{w \in V(G) | d(v, w) \leq d\}$ be the ball around vertex v with radius d. Let further $\partial B(v, d) = B(v, d) - B(v, d-1) = \{w \in V(G) | d(v, w) = d\}$ be the boundary of

(a) first extension (b) second extension

Figure 5.7: Recursive construction of 1-cop-win octile connected graphs.

the ball around v with radius d.

Let v_r and v_c be the vertices of the robber and cop, respectively. Then, after d moves of robber and cop they can reach vertices in $B(v_r, d)$ and $B(v_c, d)$, respectively. It is easy to show with proof by contradiction that if there is a vertex v_d in $B(v_r, d)$ that is not in $B(v_c, d)$ then there is a vertex v_{d-1} that is in $B(v_r, d-1)$ but not in $B(v_c, d-1)$. Hence, via induction, we know that if there is such a vertex v_d there is also a path to it that the cop cannot intersect no matter where he chooses to go. Therefore, we can reformulate (5.1) to

$$\text{TrailMax}(v_r, v_c) = \max\{2d+1 \mid B(v_r, d) \not\subseteq B(v_c, d)\} = \min\{2d-1 \mid B(v_r, d) \subseteq B(v_c, d)\}.$$

It is clear that TrailMax(v_r, v_c) is a lower bound on the value of the game when the cop starts. This is due to the fact that the robber can play according to a path p_r that fulfills (5.1) and is guaranteed to achieve a payoff of TrailMax(v_r, v_c). Therefore, it remains to show that it is also an upper bound, i.e. that in 1-cop-win octile connected maps there is a strategy for the cop such that the robber cannot achieve a higher payoff than TrailMax(v_r, v_c).

The game starts in v_r and v_c. We will be constructing a cop strategy that wins against any robber in at most TrailMax(v_r, v_c) steps. Therefore, let the robber move according to some strategy and let $r(d)$ denote his position after d steps, i.e. after d cop and robber moves. Let $\perp_d$ denote the set of vertices of $\partial B(v_c, d)$ that have shortest distance to $r(d)$, i.e.

$$\perp_d = \{\, w \in \partial B(v_c, d) \mid d(r(d), w) = \min_{w' \in \partial B(v_c, d)} d(r(d), w') \,\}.$$

Since G is 1-cop-win we know that there are no "obstacles" in the map, i.e. it can be interpreted as one big deformed open area with octile connections. This implies that $\perp_d$ is connected and forms a line segment, due to the structure of $\partial B(v_c, d)$ which is informally speaking locally rectangular shaped. This is proved in Lemma A.1 in the appendix.

We will now prove that no matter where the robber chooses to move, the cop can always respond and stay in the middle of $\perp_d$. This means that after d steps the cop will always be on $\partial B(v_c, d)$ and is in the vertices of ∂B that have minimal distance to the robber. Hence, the robber will be caught once $B(v_c, d) \supseteq B(v_r, d)$.

Without loss of generalization we can assume that $\perp_d$ forms a horizontal line segment (otherwise the proof is analog by switching the two coordinates). Furthermore, we can assume that the robber moves to the right. Consider Figure 5.8 for a depiction in an open area. The following holds for any 1-cop-win octile map. Let $a(d)$ and $b(d)$ be the two endpoints of $\perp_d$. The robber moves from $r(d)$

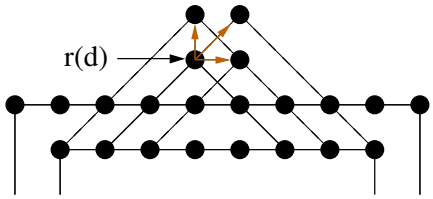


Figure 5.8: Visualization of how $\perp (r(d), d)$ can change to $\perp (r(d+1), d+1)$.

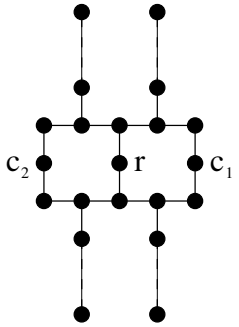


Figure 5.9: TrailMax can be arbitrarily wrong when two cops are involved.

to $r(d+1)$. Then, from $a(d)$ and $b(d)$ to $a(d+1)$ and $b(d+1)$, the horizontal coordinates of both endpoints together differ by at most two. Hence, the horizontal coordinate of the middle points of $\perp_d$ and $\perp_{d+1}$ differ by at most one. The vertical coordinates in $\perp_d$ and $\perp_{d+1}$ differ by exactly one. Hence, there is a valid move for the cop to go from the middle point of $\perp_d$ to the middle point of $\perp_{d+1}$. Note that if there is two vertices that are in the middle of $\perp_d$ the above shows that the cop can go from one of the middle vertices of $\perp_d$ to one of the middle vertices of $\perp_{d+1}$. Hence, the robber can be caught after $\min\{d \mid B(v_r, d) \subseteq B(v_c, d)\}$ moves of the cop. This completes the proof. □

This theorem also holds when the cop is faster as described at the beginning of this chapter. Let s be the speed of the cop. Then, we can redefine $\perp_d$ to be

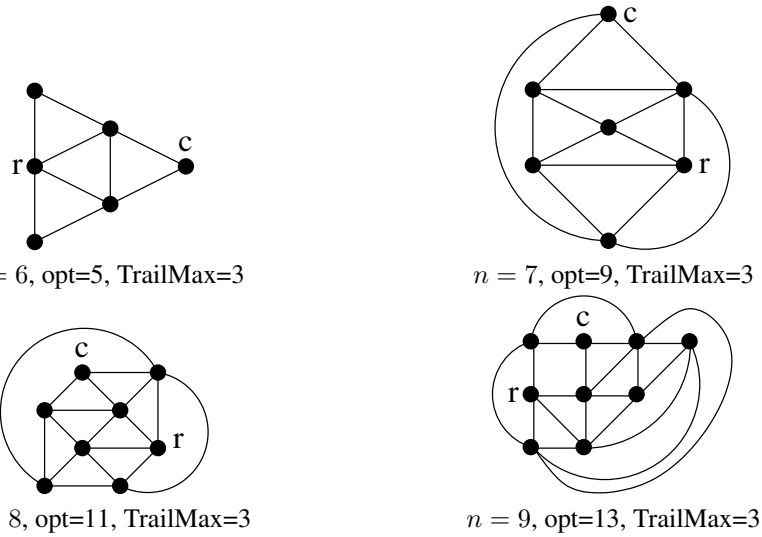
$$\perp_d = \{\, w \in \partial B(v_c, s \cdot d) \mid d(r(d), w) = \min_{w' \in \partial B(v_c, s \cdot d)} d(r(d), w') \,\}$$

and analog to the above proof it follows that the cop can move from a middle vertex of $\perp_d$ to a middle vertex of $\perp_{d+1}$.

Unfortunately, the theorem does not hold for general 1-cop-win or k-cop-win graphs ($k \geq 2$). However, it can be hypothesized that TrailMax approximates the optimal solution with some constant factor of approximation. In general, we know that TrailMax can be arbitrarily wrong for arbitrary 1-cop-win graphs. This is due to a construction of Hahn *et al.* [32] which constructs for every $T \geq 1$ a finite diameter two chordal-graph on which the robber can survive for at least T moves. Since TrailMax is bounded by the diameter (the minimization in (5.1) yields at most the diameter) this shows that TrailMax is not a constant factor approximation on several 1-cop-win graphs. The same holds true when extending TrailMax to two cops. Consider Figure 5.9. This graph is 2-cop-

vertices	2	3	4	5	6	7	8	9
Q on cop-win graphs	1	1	1	1	$\frac{5}{3}$	3	$\frac{11}{3}$	$\frac{13}{3}$

Table 5.1: Maximum of quotients of optimal value over TrailMax for all graphs with fixed number of vertices and all initial positions in these graphs.



$n = 6$, opt=5, TrailMax=3

$n = 7$, opt=9, TrailMax=3

$n = 8$, opt=11, TrailMax=3

$n = 9$, opt=13, TrailMax=3

Figure 5.10: Example graphs that yield the values in Table 5.1.

win and the length of the game can be arbitrarily long since the robber can guarantee to reach one of the four runoffs when the cops move towards him, otherwise he remains on his current position. However, when calculating TrailMax, wherever the robber goes, he can be caught by the cops if they would know where he plans to go. Therefore, the robber will remain in his current vertex and wait for the cops to capture him. Hence, TrailMax$(r, (c_1, c_2)^\mathsf{T}) = 7$.

Although we have shown that TrailMax is not a constant factor approximation, it is an interesting question if there is a function, dependent on the number of vertices in the graph, that can be used to bound the error. Given all cop-win graphs with a fixed number of vertices we computed the maximal quotient of the optimal value of the game starting in (v_r, v_c) and TrailMax(v_r, v_c) for all v_r and v_c, i.e.

$$Q(n) = \max_{G, |V(G)|=n} \max_{v_r, v_c \in V(G)} \frac{\text{opt}(v_r, v_c)}{\text{TrailMax}(v_r, v_c)}.$$

The obtained values for graphs with up to nine vertices can be found in Table 5.1. Sample graphs together with their initial position that obtain these values can be found in Figure 5.10. Note that all these graphs are planar. Unfortunately, the complete answer to this question remains for future investigation.

TrailMax can be used to generate move policies for the robber. For simplicity, the resulting algorithm will be referred to by the same name. Furthermore, a pair (p_r, p_c) for which the TrailMax equation (5.1) is maximal will be called a *TrailMax pair*. The algorithm computes a TrailMax pair (p_r, p_c) and then follows the robber's path p_r for m steps ($m \geq 1$) disregarding the cop's

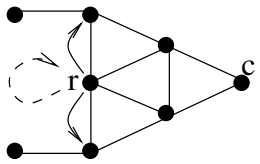


Figure 5.11: Smallest 1-cop-win graph where the set of moves according to TrailMax (solid) diverges from the set of optimal moves (dashed).

actions. Afterwards, TrailMax is called again and a new path p_r is computed, hence our notation TrailMax(m). Although we know that TrailMax is not a correct measure for the value of the game, it is possible to hypothesize that move policies generated by TrailMax(m) might yield an optimal strategy for general one or k-cop-win graphs since the solution to (5.1) is recomputed after at most m steps. Unfortunately, this hypothesis is not true. Depicted in Figure 5.11 is an example of a 1-cop-win graph where the robber is to move and the optimal move is to remain on his current position, marked with a r. This causes the cop to commit to a direction, after which the robber can run away more effectively. However, according to TrailMax, the robber has to move to either of the indicated adjacent positions.

A TrailMax pair is efficiently computed by computing $\min\{2d-1 \mid B(v_r,d) \subseteq B(v_c,d)\}$, i.e. by simultaneously expanding vertices around the robber's and cop's position in a Dijkstra-like fashion. Pseudo code of the algorithm's implementation can be found in Figure 5.12. The algorithm returns TrailMax(r,c) and a path p_r that achieves this value in (5.1). Two priority queues are maintained that store a vertex and the gcost at which the vertex has been encountered, one for the cop and one for the robber. Additionally, the robber_queue stores information from where the node has been explored, i.e. the parent. After pushing the robber's and cop's position onto the queues (Lines 2 and 3, Figure 5.12), the algorithm keeps expanding nodes from the queues. Since we compute a move policy for the robber, all nodes of a given cost for the robber are expanded first (Line 8), because the robber moves immediately after the computation. Node expansions for the robber are checked against the cop's expanded nodes to test whether the cop could have already reached that vertex and captured the robber (Line 20). More precisely, when taking a node from the queue for the robber, it is checked whether its parent's position or the node's position have been expanded, thus captured, by the cop in a previous turn. If it has been captured it is discarded. Otherwise, the vertex is declared as robber cover and expanded normally. When taking a node from the queue for the cop, it is always expanded normally (Lines 32 and 33). Moreover, if its position has not been expanded by the robber, it is declared cop cover (Lines 28 to 31).

We stop iterating when the robber's priority queue is empty. Note that since the cop explores, in the limit, all nodes of the graph and in particular all nodes that are expanded by the robber, it is always the robber's queue that empties first. However, after the robber's priority queue is empty, we know that the robber cannot reach any more nodes but it is not yet clear at which time the cop can capture the remaining nodes that have been expanded by the robber but not by the cop. This is

```
    function TrailMax( r, c )
1     if( r = c ) return (0, empty path);
2     push node on cop_queue with v ← c, gcost ← 0;
3     push node on robber_queue with v ← r, gcost ← 0;
4     number_robber_positions_caught ← 0;

5     while( robber_queue not empty )
6       qrobber ← node with best gcost on robber_queue;
7       qcop ← node with best gcost on cop_queue;
8       if( qrobber.gcost ≤ qcop.gcost )
9         expand_robber_node( qrobber );
10      else
11        end_iteration ← expand_cop_node( qcop );
12        if( end_iteration ) break;

13    if( number_robber_positions_caught < size of robber_closed )
14      while (cop_queue not empty)
15        qcop ← node with best gcost on cop_queue;
16        end_iteration ← expand_cop_node( qcop );
17        if( end_iteration ) break;

18    return generate_path();
```

```
    procedure expand_robber_node( qrobber )
19    take qrobber from robber_queue;
20    if( qrobber.v not in robber_closed and qrobber.v not in cop_closed and
          qrobber.parent not in cop_closed )
21      insert new entry into robber_closed with v ← qrobber.v, parent ← qrobber.parent, gcost ← qrobber.gcost;
22      for all w ∈ N(qrobber.v)
23        push node on robber_queue with v ← w, gcost ← qrobber.gcost + c(qrobber.v, w), parent ← qrobber.v;
```

```
    function expand_cop_node( qcop )
24    take qcop from cop_queue;
25    if( qcop.v not in cop_closed )
26      insert qcop.v into cop_closed;
27      last_cop_turn ← qcop.gcost;
28      if( qcop.v in robber_closed )
29        number_robber_positions_caught ← number_robber_positions_caught + 1;
30        last_caught_vertex ← qcop.v;
31        if( number_robber_positions_caught = size of robber_closed ) return true;
32      for all w ∈ N(qcop.v)
33        push node on cop_queue with v ← w, gcost ← qcop.gcost + c(qcop.v, w);
34    return false;
```

```
    function generate_path()
35    result_path ← empty path;
36    rce ← entry in robber_closed belonging to last_caught_vertex;
37    for i ∈ [rce.gcost, last_cop_turn)
38      add last_caught_vertex to the end of result_path;
39    while( rce.gcost ≠ 0 )
40      add last_caught_vertex to the end of result_path;
41      last_caught_vertex ← rce.parent;
42      rce ← entry in robber_closed belonging to last_caught_vertex;
43    reverse result_path;
44    return (2*last_cop_turn, result_path );
```

Figure 5.12: Pseudo code implementation of TrailMax. All functions and procedures share the same data structures.

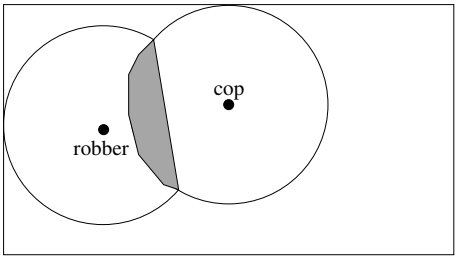


Figure 5.13: Visualization of TrailMax's computation. The gray area is the nodes that have been reached by the robber first, declared as robber cover but will not be expanded anymore since they were captured by the cop in a previous turn.

why we have to keep iterating until eventually all robber vertices are captured (Line 13 to 17). The last node that is captured is the goal node where the robber will run to (last_caught_vertex in Figure 5.12). Path generation is done by tracing back the predecessor information in the robber's closed list (Lines 39 to 42). The path has to be extended by waiting actions, i.e. remaining on the current vertex, until the cop captures the robber (Lines 37 and 38).

A visualization is depicted in Figure 5.13. The gray area indicates the vertices that are declared robber cover but are not expanded anymore since the expansion around the cop's position captured them in a previous turn. Computation ends when all nodes declared as robber cover have been expanded by the cop as well. The last node that is captured by the cop is the goal node the robber will run to.

As a by-product of the TrailMax computation, we also compute the robber's cover as defined in our version of the Cover heuristic. In contrast, the advantage of TrailMax is that we only have to compute once and extract a path that maximizes the length of survival. The Cover heuristic has to be computed for every possible move an agent could take and only returns information about the size of the area he can reach before any other agent.

Note that for a naive implementation to compute TrailMax we would have to keep reexpanding nodes in the interior of the Dijkstra region for the robber to explore all possible paths he could take. However, the above computation finds a goal vertex and only a shortest path to it that is extended by making the robber remain on his goal vertex until capture. It is not hard to see that this extended shortest path is indeed a solution to the TrailMax equation (5.1) and hence the computation is correct. Assume that there is a path p_r starting in the current robber's vertex r that can only be intersected by the cop, starting in c, at a vertex g. Additionally, assume that the cop can intersect the shortest path from r to g in a vertex f. We depicted this situation in Figure 5.14. Furthermore, we denote the path from a vertex v to w as p_{vw}. Since p_r is not a shortest path we have cost$(p_r) >$ cost(p_{rf})+cost(p_{fg}). Since the cop can intersect the shortest path from r to g we also have cost$(p_{rf}) \geq$ cost(p_{cf}) and therefore cost$(p_r) >$ cost(p_{cf}) + cost(p_{fg}). Hence, the cop can reach g before the robber. By moving from g in the opposite direction of p_r he can then use his time to move towards the robber and capture the robber before he reaches g. This is a contradiction to the assumption. Thus, for

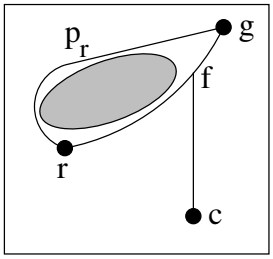


Figure 5.14: Finding the shortest path and extending it by waiting actions is a solution to the TrailMax equation (5.1).

every path p_r the shortest path from the start to the intersection vertex extended by remaining on the intersection vertex yields the same value as p_r in the TrailMax equation (5.1). This shows that our TrailMax computation is correct.

Note that there might be many possible goal vertices the robber could run to and many different paths to get to them that fulfill the TrailMax equation (5.1). Finding all such vertices is possible by remembering all robber nodes that have not been caught before the last cop's turn expansion. This could potentially be used to take advantage of a suboptimal cop, although we do not study this issue here.

Within computer game maps, edge costs are often approximated to enable faster computation. Under the assumption that path costs can only differ by a fixed number of values, buckets can be used within the priority queue and queue access takes constant time. Assuming a bounded degree of the graph the above algorithm runs in time linear in the size of the graph.

Although TrailMax already scales well to large maps (cf. Section 5.5) our goal is to make computation time as independent of the size of the input graph as possible. Inspired by DAM we use abstraction to achieve this goal. Pseudo code can be found in Figure 5.15. Let d be the number of levels of abstraction where the robber and the cop occupy distinct vertices. Then, computation starts on level $\ell \leftarrow d/2$ (Line 2, Figure 5.15). On each level of abstraction a TrailMax solution is computed. If the solution length does not exceed a certain value q (Line 5), then computation proceeds to the next lower level. If it does, the computed abstract path is refined to a ground level path using PRA*'s refinement (Line 7), i.e. progressively computing a path on the next lower level that only goes through nodes whose parents are either on or adjacent to the abstract path. In the following, this algorithm is called *Dynamic Abstract TrailMax* with threshold q and number of steps the solution is followed m, hence DATrailMax(q, m).

In modern computer games players have the ability to change the world by their actions. That means that the graph that the game is being played on might change dynamically. For instance, consider one player unlocking a door in the world and therefore establishing a connection between before disconnected components. Such changes can be handled efficiently by computing a new move policy, i.e. a new solution path. Since our algorithms compute a new move policy every k

```
    function datrailmax( r, c, q )
1       compute abstract robber and cop locations r_i and c_i;
2       compute start level ℓ;
3       for level from ℓ to 1;
4         (t, p) ← trailmax( r_level, c_level );
5         if( t ≥ q ) break;
6       if( t ≥ q )
7         refine path p to ground level;
8       else
9         (t, p) ← trailmax( r, c );
10      return p;
```

Figure 5.15: Dynamic Abstract TrailMax pseudo code implementation.

steps they adopt to new scenarios quickly.

Another problem faced with computer games is that the entire graph might not be known. The robber might have limited visibility and might only know a part of the graph. Furthermore, he might not "see" the cops if they are beyond his visibility window. Here, our algorithms can be used by making the assumption that the unknown parts of the map are free space and hence directly traversable. Then, we can compute TrailMax or DATrailMax solutions solely on the visibility window. Thus, these two algorithms are suitable for dynamic computer games with visibility restrictions.

5.5 Experiments

5.5.1 Suboptimality

To evaluate all the previously discussed algorithms we measure the quality and required computation time in terms of node expansions. We set $d = 2$, i.e. the cop can take two turns before the robber gets one and can thus move to any location within a radius of 2 around his current location. First experiments show that even greater cop speeds yield the same trends. In contrast, since capture occurs faster, the game becomes easier and less interesting for the robber.

To generate meaningful statistics we use four sets of maps. All maps are taken from the commercial game Baldur's Gate.

set	number of vertices	number of maps
one	175 - 996	22
two	1,014 - 1,999	29
three	2,037 - 4,690	26
four	5,201 - 22,841	22

A plot of a sample map from the third set can be found in Figure 5.16. Furthermore, 1000 initial positions for each map are generated randomly. We choose the selection at random because we want to explore the performance of the algorithms for all scenarios since in a video game, both agents could potentially be spawned anywhere in the map.

We choose octile connections for the map representation and subsequent levels of abstraction are generated using Clique Abstraction [76]. To enable effective transposition table lookups in Minimax

Figure 5.16: One of the maps used in Baldur's Gate that the experiments were conducted on. The gray parts are traversable.

and DAM we set all edge costs to one in all levels of abstraction. Thus, the distance metric between two positions (on an abstraction or ground level) becomes the maximum norm metric between these positions (cf. d_∞ in Section 4.5). Furthermore, equidistant edge costs mean we are optimizing the number of turns both players take rather than the distance they travel. All the tested algorithms can be used for non-equidistant edge costs, only Minimax's and DAM's performance is expected to be lower.

Using the RA algorithm from Chapter 3 the entire joint state space is solved first, i.e. we compute the values of an optimal game for each tuple of positions of the robber and cop. This is done in an offline computation and is used to generate optimal move policies for the cop as well as to know the optimal value of the game. The time needed to compute these offline solutions is depicted in Figure 3.5(c).

We study the following target algorithms:

Original Cover. The target performs hill climbing due to the original Cover heuristic (cf. Section 5.3). The heuristic has to be computed in every step and for every possible move. We ported the implementation of Isaza *et al.*[36] into our framework.

Improved Cover. We test our redefinition of Cover outlined in Section 5.3.

Greedy. The target performs hill climbing using distance heuristic d_∞. This is extremely fast since distance evaluation is very simple. We also experimented with hill climbing due to the perfect distance heuristic d_{perfect}, which we call improved Greedy. The perfect distance heuristic is precomputed in an offline step before the evaluation.

Minimax. The target runs Minimax with α-β pruning, transposition tables and the admissible forward heuristic from Section 4.5 with underlying distance heuristic d_∞. We experimented with depth bounds from 1 to 11.

DAM. The target runs the original Dynamic Abstract Minimax algorithm as described in Section 5.2. The computation starts at fractions $1, \frac{3}{4}, \frac{1}{2}, \frac{1}{4}$ of the abstraction hierarchy where both players occupy distinct nodes. Furthermore, we experiment with minimax computation depths from 1 to 11. We indicate this with DAM(q, d) where q is the fraction and d is the depth.

DAM with one step refinement. The target runs Dynamic Abstract Minimax where only one ab-

stract step is refined into the lower levels. Analogously to DAM we indicate this algorithm by DAM1(q, d).

RandomBeacons(1-20). We tested RandomBeacons(b,m) for $b = 40$ and $m = 1, \ldots, 20$ (cf. Section 5.1).

TrailMax(1-20). We tested TrailMax(m) for $m = 1, \ldots, 20$ (cf. Section 5.4).

DATrailMax(10,1-20). We tested DATrailMax(q,m) (cf. Section 5.4) for $m = 1, \ldots, 20$. $q = 10$ was chosen by hand. The question whether there is a better setting remains for future investigation.

To evaluate performance the game is simulated for each initial position on each map. Within these simulations, the target algorithm is called whenever a new move has to be generated. TrailMax, DATrailMax and RandomBeacons are only called when a new path has to be computed, thus the number of turns and algorithm calls differ in this case. All other algorithms are called once per turn and therefore these two numbers are equal.

In fact, it is not possible for TrailMax, DATrailMax and RandomBeacons to spread their computation among the turns where the previous computed path is followed because the future position of the cop is unknown. Nonetheless, when used in computer games, these algorithms will only require computation once every m steps and therefore make the frames during path execution available to other tasks. This is the motivation to analyze the computation time per turn for these three methods.

We are interested in the following performance measures:

- the expected survival time of the target measured in percentage of the optimal survival time (suboptimality),
- the number of nodes expanded per call (nE/c) and nodes touched per call (nT/c) to the algorithm within one game simulation, and
- for TrailMax, DATrailMax and RandomBeacons the amortized number of nodes expanded per turn (nE/t) and nodes touched per turn (nT/t) within one game simulation.

To account for variable sized maps, the numbers of nodes expanded and touched are further normalized and measured as a percentage of the map size (the number of vertices in the map). Nodes expanded counts how many times the neighbors of a node were generated, while nodes touched measures how many times a node was visited in memory.

The results are in Table B.1 and the values from the fourth set are plotted in Figure 5.17. Note that each algorithm is plotted multiple times in Figure 5.17 for different parameter settings. The x-axis is reversed so the best algorithms are near the origin, with high optimality and few expansions per move. Notice further the logarithmic scale on the number of node expansions. A Pareto-optimal boundary is formed by the Greedy, TrailMax and DATrailMax algorithms, meaning that all other algorithms have either worse optimality or more node expansions per turn, on average.

Concerning computation time, the Cover algorithm clearly performs the worst. Having to compute the heuristic in every step and for every possible move, its computation time is beyond any

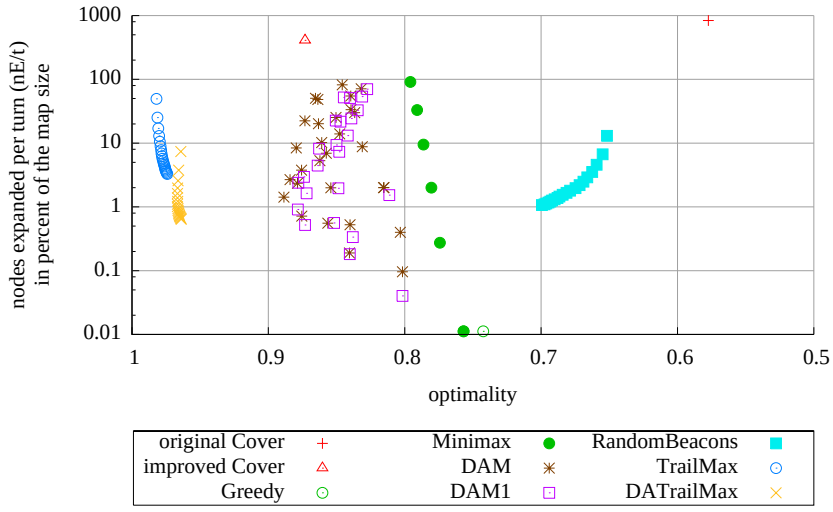


Figure 5.17: Optimality versus node expansions per turn in one game simulation. Data is taken from the fourth set, i.e. maps with at least 5000 vertices. Averaged over the number of games played in the experiments. Left bottom corner is best, right upper corner is worst.

computer game requirement. Although our improved version of Cover expands fewer nodes than the original version, since we stop expanding nodes when the robber's priority queue becomes empty, it touches more nodes because the expansion for the cop is not stopped whenever it reaches the area already covered by the robber. Abstractions can be used to compute solutions in less time, but optimality decreases when using abstract solutions since the heuristic is most informed at the base level. Since both algorithms are not the most optimal approaches, even without using abstractions, we did not consider using abstractions here. Note that the computation of our improved Cover definition is very similar to the computation of TrailMax, the difference being that Cover optimizes for the size of the area in which the robber can run away, whereas TrailMax optimizes for the time the robber can survive in this area. It is evident that our redefinition significantly improves optimality with respect to the original Cover algorithm.

Considering quality, RandomBeacons is the second worst algorithm. This is due to the fact that it does not play very well in the endgame, i.e. when the target is cornered and is about to be captured. When distributing the beacons, many of them lie in parts of the map that are heuristically far away from the cop. Thus, the robber runs towards these positions. Since he is cornered, this results in running into the cop. Although the algorithm is very simple, the number of node expansions and nodes touched is relatively high. This is caused by PRA* that is used to find a path to the best beacon.

As expected, Minimax becomes closer to optimal when the depth is increased. Geometrically, the computation depth is (twice) the radii of circles around the cop's and robber's position. Each pair of vertices within these circles, respectively, is a leaf node in the game tree that is evaluated by Minimax (recall that we do not search the entire tree due to α-β pruning). Hence, when increasing

the search depth we increase the radii of these circles, i.e. Minimax includes more information about the surroundings of each agent. However, its computation time increases exponentially. When using a depth of seven and greater it already expands more nodes per call than DATrailMax.

Abstract levels have cycles in them and minimax can find how to exploit such cycles even with shallow searches. Hence, DAM's computed strategies on abstract levels are similar for different search depths. Therefore, DAM does not significantly increase in optimality when its depth parameter is increased.

It is surprising that Greedy, i.e. hill climbing with maximum norm distance metric, performs extremely well. Due to the fact that this algorithm requires almost no computation time, we can conclude that Greedy is the method of choice when optimality is of minor importance. Furthermore, from Table B.1 we can see that its performance is relatively stable over all sets of maps. Unfortunately, the immediate assumption that Greedy will do better in the two-player game when a more accurate distance metric, i.e. single agent heuristic, is used, does not hold. This is due to the fact that the improved Greedy algorithm, which uses a perfect distance metric, does not perform significantly different with respect to optimality (cf. Table B.1). In addition, this also shows that the game of cops and robber is not trivial. Even though we have a solution to the single agent game, pathfinding, this solution is not very optimal for the adversarial game, i.e. minimize/maximize the time the distance of cop and robber is greater than zero.

TrailMax and DATrailMax perform best with respect to optimality. Although DATrailMax uses TrailMax on abstract levels it experiences only a small reduction in optimality. On the contrary computation time decreases drastically. For instance, on the fourth set DATrailMax expands about 6 times fewer nodes per call than TrailMax. Notice that, although the computation time per call is fairly high, the amortized time per turn is small and even comparable to RandomBeacons. Note that on the smaller maps, DATrailMax expands and touches a higher percentage of nodes. This is because the abstraction is not as useful and therefore the algorithm degenerates into TrailMax.

While Figure 5.17 shows the averaged points of all game simulations, the actual results are clouds of points where each point represents the performance in one game. We compare this underlying data for the two best algorithms, TrailMax and DATrailMax, in Figure 5.18. The light points contain data for DATrailMax(20,10), while the dark circles are the data points for TrailMax(20). The x-axis is reversed and the y-axis is logarithmic. DATrailMax is clearly faster. Trailmax has a slight advantage in the number of times it makes optimal moves, resulting in slightly better optimality. Notice that although there are games where both algorithms perform poorly with respect to optimality, most reside at above 80% of optimality. Furthermore, node expansions for both algorithms are uniformly bounded at around 7% of the size of the map.

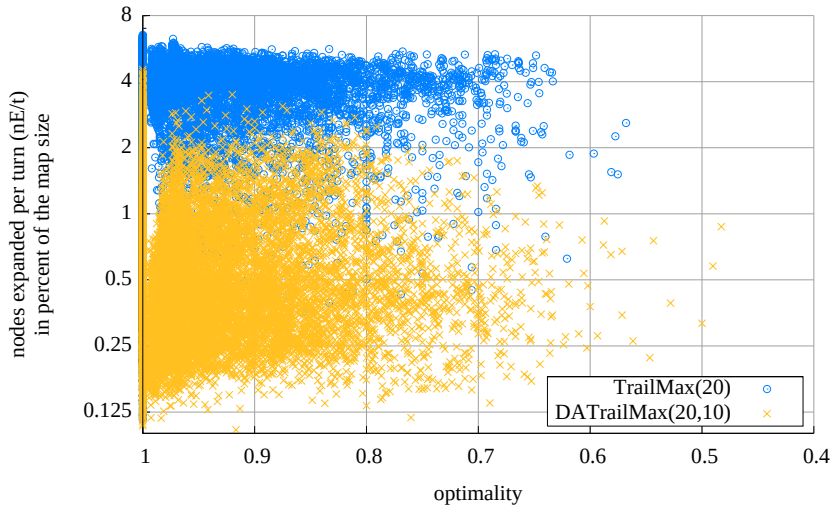


Figure 5.18: Optimality versus node expansions per turn in one game simulation. Plotted for all games played in the experiments. Left bottom corner is best, right upper corner is worst.

5.5.2 Exploitability

The above experiments evaluate the algorithms' performances against a cop that plays according to a Nash equilibrium. Hence, the cop makes the assumption that the robber will play optimally. This can be thought of as evaluating the suboptimality of the robber algorithm.

In this subsection we will evaluate the performance of the algorithms if the cop is given the robber's strategy. Thus, we compute best responses against the robber algorithms. Given a deterministic robber strategy that does not keep track of the history of the game, we can use the RA algorithm from Section 3.1 to compute a best response. However, not all of the previously presented algorithms are deterministic and do not track the history of the game. Therefore, we cannot evaluate the full spectrum of algorithms of the last subsection. Nevertheless, since the performance of the algorithms tested here against a best response is similar to the performance against an optimal cop we can conclude that the performance of the untested algorithms will be similar to their performance in the previous subsection.

Computing a best response requires solving the game, i.e. computing a value for every pair of initial positions. Furthermore, in the computation of the RA algorithm, the successor relation for the robber is substituted by the choice of the given algorithm. Hence, we have to compute the algorithm's move for every non-terminal state in which the robber is to move. This significantly increases computation time with respect to the RA algorithm. The increase is dependent on the computation time requirements of the given robber algorithm and thus varies widely (see Table 5.2).

To enable termination of the experiments in feasible time, we only computed best responses for the first set of maps used in the previous subsection. The other sets induce larger maps, hence much larger state spaces and thus impractical computation times. Furthermore, we only compute a best

algorithm	exploitability		suboptimality	max. computation time for a map
original Cover	37.81%	(62.18%)	69.62%	358 min
Greedy/Minimax(1)	31.77%	(68.23%)	79.74%	13 s
Minimax(5)	27.05%	(72.95%)	85.95%	10 min
DAM(0.5,3)	20.47%	(79.53%)	90.12%	135 s
DAM1(0.5,3)	17.37%	(82.63%)	92.43%	125 s
improved Cover	9.99%	(90.01%)	92.75%	1067 min
DATrailMax(1,10)	2.32%	(97.68%)	98.83%	72 min
TrailMax(1)	1.66%	(98.34%)	99.11%	152 min

Table 5.2: Averaged exploitabilities for the first set of maps.

response for some of the algorithms in the last subsection. More precisely, we took one candidate for each algorithm and chose the parameters with which the algorithm performed best in the previous section.

Exploitability means how much a knowing cop can take advantage of the respective given target algorithm. To evaluate this term we first compute the optimal values of the game beginning in every state. Then, we compute a best response, i.e. compute the payoff the robber can achieve against a cop that plays a best response against him, for every state. We compute the quotient of the best response payoff over the optimal value of the game. Informally speaking this is the achieved suboptimality against a best response cop. The exploitability is the difference between one and the average of this quotient over all non-terminal states where the robber cannot be caught immediately by the cop, i.e. he gets to move at least once in a game starting in the respective state.

We plot the exploitability of the algorithms for each map in the first set of maps in Figure 5.19. These values are averaged in Table 5.2. Furthermore, we report the maximal, over the set of maps, computation time needed to compute a best response on a map.[1] To compare these results with the algorithms' achieved suboptimalities we include the above quotient and the suboptimalities from the previous subsection. It can be seen that these values resemble the previous experiments. The ordering of the algorithms due to suboptimality is exactly the same as the ordering due to exploitability. This demonstrates that computing a Nash equilibrium and using it for the cop's strategy is a good approach. This is because computing a best response is much more expensive in computation time but does not yield much gain against the outlined algorithms. Furthermore, the global approaches Cover and the TrailMax variants do not have obvious flaws that can be largely exploited. Otherwise, the reported quotient, i.e. achieved suboptimality against a best response cop, would significantly differ from the actual achieved suboptimality against a Nash equilibrium playing cop. In contrast, a target running local approaches like Minimax, that only has a very small lookahead, and consequently DAM, that uses Minimax on abstract levels, is more exploitable when its strategy is known to the cop.

[1] All experiments were run on a Intel Xeon© 2.5GHz CPU with 16GB RAM.

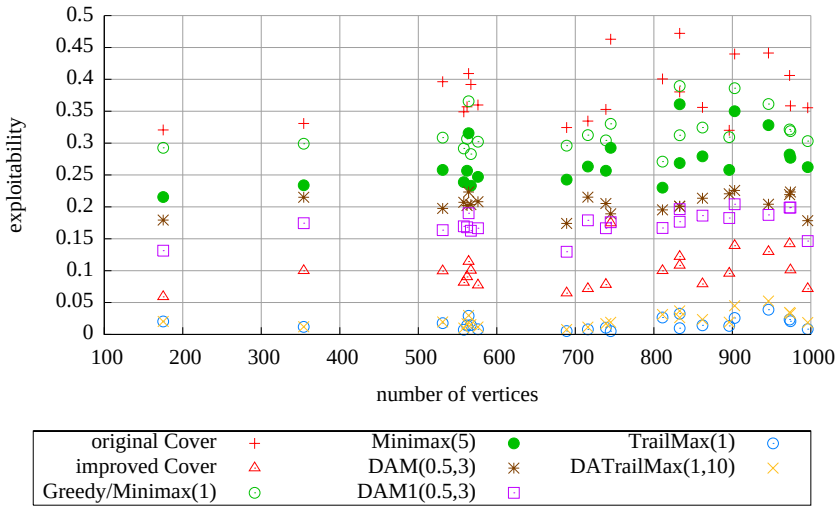


Figure 5.19: Exploitability of all the algorithms with respect to the map size for the first set of maps.

5.5.3 Comparison to original Cover framework

When running our experiments we discovered an ambiguity. Isaza *et al.* [37] claim very good results when Cover with Risk and Abstraction (CRA) is used for the cops. Furthermore, Isaza's thesis [36] states that the original Cover heuristic performs well when used for the target side. In contrast, our results clearly demonstrate that the original Cover algorithm is not a good algorithm to use for the robber. This is due to multiple reasons that we will outline in the following.

We are the first to conduct a consistent study that evaluates Cover's suboptimality and how much it can be exploited. Isaza *et al.* [37] used a best response against a target cover algorithm in their analysis but only on three very small maps with 20 initial positions each. Since the maps are small and only a few instances were computed these experiments do not report the true characteristics of the Cover heuristic. Moreover, they evaluated all their cop algorithms against specific target algorithms, neither including best responses nor an optimal moving target. Therefore, their performance measures of Cover rely on the performance of other approximations and neither an optimal or best response opponent.

Isaza's thesis [36] shows experiments on five graphs where optimal strategies are computed and compared to CRA (for the pursuers) and hill climbing with Cover heuristic (for the target). Informally, all these graphs have the property that maximizing the covered area of the cops also minimizes the capture time. Analogously, when the robber wants to maximize capture time he has to counteract the cops, hence maximize his cover to minimize theirs. Therefore, Cover naturally performs well on these graphs. On general computer game maps, these properties do not hold and our experiments show that Cover's performance is significantly different from Isaza's observations.

To support our results, i.e. to show that the deviations in results are not caused by our implementations, we also conducted experiments in the framework used by Isaza. In the following

we will first outline the differences between ours and their movement model. Second, we discuss how optimal strategies from our framework can be converted into Isaza *et al.*'s framework and why we cannot compute optimal strategies directly within their environment. Third, we will show that objective values in the two models resemble each other which indicates that the differences in movement are minor. Fourth, we conduct experiments in their framework which show that the original Cover algorithm does perform similar to our suboptimality experiments in Subsection 5.5.1. Additional experiments with multiple cops show that Cover's performance is invariant of the number of cops. Finally, we give examples of cop-win graphs where Cover moves suboptimally, disproving a theorem from Isaza's thesis that claims that Cover is optimal on cop-win graphs.

Their framework uses edge costs as the time it takes to take an action. The agents can declare new actions whenever they complete their previous actions. Speed is also modeled differently by dividing the time it takes to take an action by the speed of the agent. This causes the agents to interleave their turns in a different way than in our model. Recall that our model is purely alternating. When an agent takes an action it is executed immediately. The cost of the action is the edge cost and the cost for a wait is one.

To run equivalent experiments in their framework it is important to notice that Isaza's perception of time is different from ours, hence the two players optimize slightly different metrics. We run our experiments with one fast cop against a slow robber. Since both players execute their actions simultaneously, although actions are chosen alternatingly and the time it takes to execute an action is the edge cost induced by that action, the time it takes to capture the robber is roughly speaking the distance the cop travels (divided by his speed). In contrast, in our model, the time to capture is the distance both agents travel.

Our experiments were run on the map depicted in Figure 5.16 and a robber with unit speed chased by a cop moving twice as fast. This map has 5672 vertices and was modeled with octile connections (cf. Definition 2.8). To account for the different player objectives in Isaza *et al.*'s framework we computed a Nash equilibrium in our framework that optimizes for just the traveled distance of the cop. More precisely, we computed the solution lengths for all pairs of possible initial positions. Furthermore, to match Isaza's framework even closer, we computed with edge costs not all equal to one. Note that this is in contrast to the previous experiments. We set the cost of a vertical or horizontal move to one and the cost for a diagonal move to $\sqrt{2}$. Recall that this actually results in a rather unnatural model (see the discussion on time at the beginning of Chapter 3).

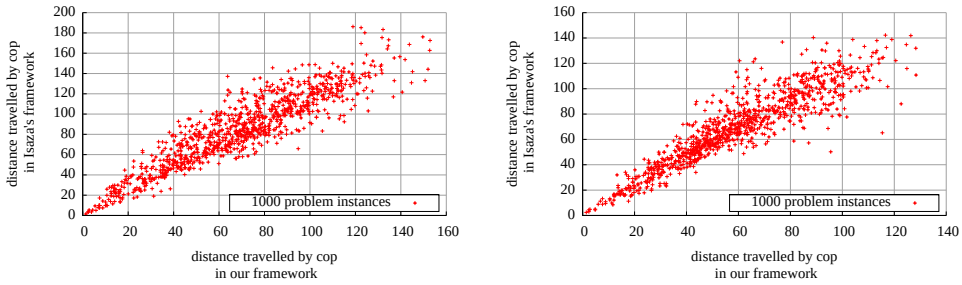
We computed the solution to the maps in our environment and transferred them into the framework of Isaza *et al.*. The strategies that use these values will be called converted optimal strategies in the following. They move along the action field induced by the game values to all possible pairs of positions. We chose the name converted optimal because these strategies are optimal in our framework. Unfortunately, they are not optimal in Isaza *et al.*'s framework. This is due to the fact that their model of octile connections is slightly different from ours: their model is the pure induced

graph of the octile connected grid, i.e. moves over corners are allowed. Furthermore their turn-taking is different because the agents declare their moves whenever execution of the previous move terminates. But, it is expected that our converted optimal strategies are a good approximation to a real Nash equilibrium of Isaza's variation of the game since they have the same objectives only with the underlying assumption of a slightly different movement model. Note, that it is very difficult to compute real optimal solutions or best responses for their framework. This is because the RA algorithm (cf. Section 3.1) and RMA* (cf. Section 4.4) are much less efficient due to how the players take their turns.

Furthermore, it is expected that a cop playing according to our converted optimal strategies can catch an arbitrary robber also in Isaza *et al.*'s framework. This is because even though the cop might make mistakes due to the different movement model, he gets to move twice during the time the robber moves once. Therefore, he is still guaranteed to move closer to the robber with every move. This was verified in our experiments where the converted optimal cop was always able to achieve capture. Hence, our computed results are actually an upper bound on the values of a real Nash equilibrium.

To compare objective values, i.e. to further support the claim that the differences in the movement model are minor and therefore our converted optimal strategies can be expected to be close to a real Nash equilibrium, we computed the distances traveled by the cop for different configurations of target and pursuer algorithms. We used the 1000 initial pairs of positions from Section 5.5 on the map depicted in Figure 5.16. We used a cop strategy that computes a path towards the current robber position via PRA* and follows it for one move before recomputing a new path. The first target used hill climbing due to the distance heuristic (see Greedy in Section 5.5). The second target used hill climbing due to the Cover heuristic. A plot of the solutions lengths, i.e. distance traveled by the cop, for these two configurations can be found in Figure 5.20 (note the different scales on both axis). This clearly shows that solution lengths resemble each other in the two frameworks. For long solutions, i.e. when many moves are required until capture, the gained values differ more since the above differences in the movement models can have more effect. Since PRA* and Greedy do not depend on the different movement models the deviations in Figure 5.20(a) are only due to the execution of the strategies with different movement models. In contrast, the Cover heuristic uses the movement models in its computation. This explains the greater deviations in Figure 5.20(b) since now in addition to the different execution the algorithms themselves decide differently in the two frameworks.

We now support our claim that Cover performs poorly when used as a target algorithm. For the above 1000 initial positions we computed the solution to the games when a converted optimal cop plays a converted optimal robber and when a converted optimal cop plays a hill climbing robber due to the Cover heuristic. This is an analogy to the previous suboptimality experiments. The first game is an approximation to the optimal value and the second game is the evaluation of cover against an



(a) PRA* cop vs. Greedy target

(b) PRA* cop vs. Cover target

Figure 5.20: PRA* cop vs. Greedy and Cover targets.

optimal playing cop.

Recall that the Cover heuristic is designed to maximize the agent's covered area. We also experimented with an opposing approach for the robber. Instead of hill climbing, the algorithm tries to decrease the heuristic value, i.e. to minimize its covered area. Since the cop covers all the area that is not covered by the robber, this approach is equivalent to maximizing the cop's cover instead of the robber's cover. We will call this method negative Cover in the following.

The results are plotted as point clouds in Figure 5.21. When averaging the obtained values we discover the following suboptimalities

	solution length in % of converted optimal solution length in Isaza *et al.*'s framework	suboptimality in our framework
Cover	64.09%	58.78%
negative Cover	60.76%	64.37%

As a comparison we have included the obtained suboptimalities of our previous experiments from Subsection 5.5.1 computed solely on the 1000 instances of the map in Figure 5.16. Note that the agents optimize for the accumulated number of turns in our framework rather than the distance traveled by the cop. These results clearly show that the performance of Cover is similar in both frameworks.

We have investigated Cover's performance when one fast cop plays a slow robber. The experiments performed by Isaza *et al.* include multiple cops at the same speed of the target or slower than the target. Although they did not perform a statistically significant study it remains to show that the original Cover algorithm will perform similar even when multiple cops are involved. Since the state space with two cops is significantly larger than the state space with only one cop we could not conduct experiments on the map used above. We used the map depicted in Figure 5.22 that has 501 vertices and was modeled with octile connections. Hence, the state space has 63,001,251 states, approximately twice as many as in the above experiment even though the map is significantly smaller. We used two cops and one robber with the same speed and computed a Nash equilibrium in our framework that optimizes for the sum of distances the cops travel.

When converting this equilibrium to Isaza *et al.*'s framework we noticed that it is easily possible to force two converted optimal cops into an infinite loop. As an example, consider a situation where

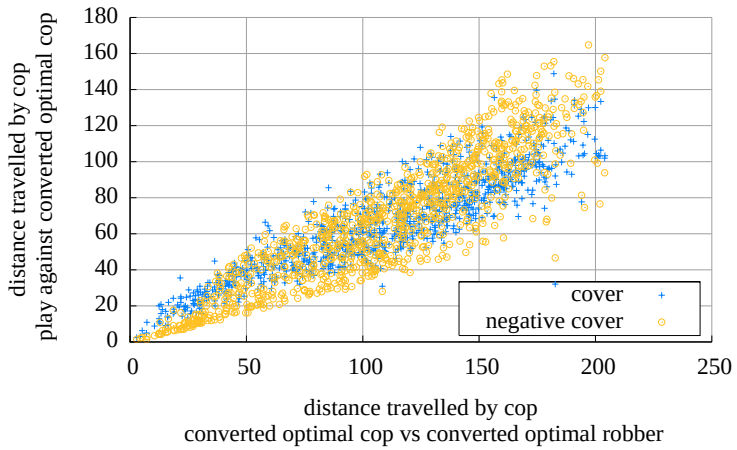


Figure 5.21: Converted optimal cop playing hill climbing due to Cover and negative Cover heuristic.

the robber is currently moving towards a vertex v, one cop is adjacent to v and both cops have to declare their move. This is interpreted as if the robber is already on v. In our framework, the cop adjacent to v would move to v, immediately capture the robber, and the second cop could stand still. Hence the first cop moves towards v and the second stays still. However, while the first cop is still moving towards v, the robber arrives in v and chooses a new vertex to move to. When the first cop then arrives in v he faces the same situation as before. This can lead to infinite move cycles that are only due to the different movement model.

Since in a video game it is of minor importance whether the opponent is exactly on top of the agent or just very close (less than one vertex away) we eliminated this problem here by declaring capture if, at any given point in time, the robber and a cop each execute a move that leads towards the same vertex. Hence, both agents do not have to arrive at the same time in this vertex but only move towards it at the same time. The Cover implementation in Isaza *et al.*'s framework requires the definition of capture. Therefore, we also modified its implementation to respect this change.

We used 1000 initial positions for the robber and the two cops on the map depicted in Figure 5.22. We measured the sum of the distances the cops traveled throughout each simulation. As an approximation to the optimal value we used the values obtained by a converted optimal robber competing against two converted optimal cops, named converted optimal solution length. Running hill climbing with Cover and negative Cover heuristic gave the following results:

	solution length in % of converted optimal solution length
Cover	69.60%
negative Cover	69.53%

Although these results are slightly better than for the one cop case this shows that Cover still performs the same since the map was smaller and did not have many big open areas that could confuse the heuristic (see the counterexamples against optimality for 1-cop-win graphs below).

Both of the above experiments also show a different problem. When the target minimizes its

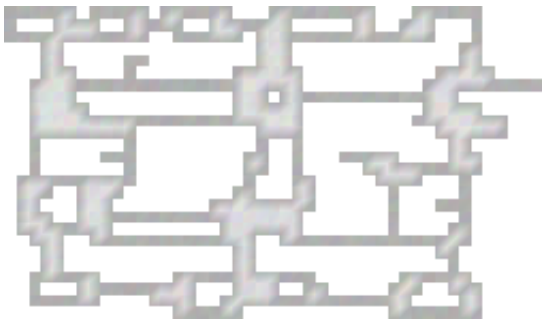


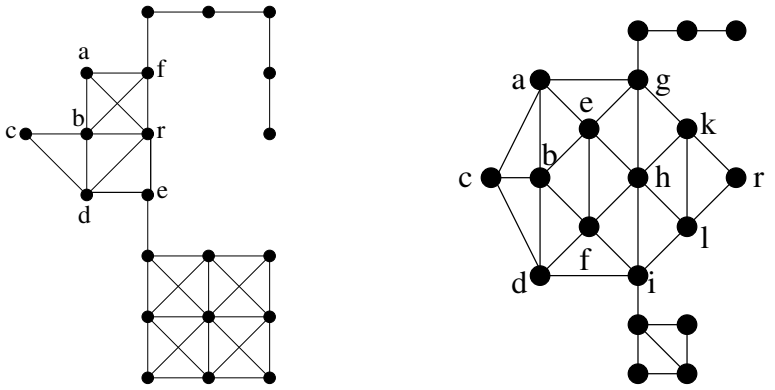
Figure 5.22: The map used to compare Cover with a converted optimal target when playing two converted optimal cops.

own covered area, i.e. negative Cover, it achieves a similar suboptimality compared to maximizing it's own covered area. The latter is usually interpreted as the robber counteracting the cops cover approach. The prior can be interpreted as the robber cooperating with the cops. Since Cover has the incentive to spread teammates across the map this effectively results in the robber trying to distance himself from the cops. However, it seems a contradiction that the robber cooperates with the cops when actually they are playing an adversarial game. This suggests a very low correlation between the covered area of an agent and its actual ability to perform well in the game.

Isaza's thesis [36] provides a proof that the cover heuristic gives the exact solution cost when used on 1-cop-win graphs. He addresses the mathematical game and optimality is meant with respect to the number of turns taken when both agents move at same speed, taking turns alternatingly. It is also implied that hill climbing due to the cover heuristic is an optimal strategy for the cop. In the following, we will outline why these facts are incorrect and show counterexamples.

Unfortunately, Isaza *et al.*'s framework implements the cover computation slightly different than described in his thesis. Figure 5.23 depicts one counterexample for both variants. It further provides some insight in why Cover is poorly correlated with playing well in the game. The underlying idea of these counterexamples is roughly to have a long path that works as a runoff for the robber and an area that has many nodes but only a small capture time. Then, we can show that the cop makes the wrong move protecting the large area from invasion by the robber which enables the robber to run away much longer due to the long runoff. Informally speaking we show that the area covered does not correlate with the actual capture time.

First consider the computation described in Isaza's thesis [36]. Here, Cover is computed beginning with the cop taking actions, regardless of who's turn it is going to be in reality. Consider Figure 5.23(a), the cop being at c, the robber at r and it is the cop's turn. The cop can decide whether to stay at c, to go to b or d. Staying in c is clearly dominated by either going to b or d, for optimality and the cover computation alike. When he moves to b he then evaluates the new position by the Cover heuristic. The evaluation starts with the cop taking action, regardless of the fact that it is actually the robber's turn. Therefore, the sequence of expansions is



(a) Cover as described in Isaza's thesis

(b) Cover as implemented in Isaza's framework

Figure 5.23: A counterexample on 1-cop-win graphs that shows that Cover is not optimal when used for the cop.

time		cop cover	robber cover
0	b is expanded by the cop	b	$\emptyset$
0	r is expanded by the robber	b	r
1	r is not expanded by the cop since already covered		
	a, c, d, f are expanded by the cop	a, b, c, d, f	r
1	a, b, d, f are not expanded by the robber since already covered		
	e is expanded by the robber	a, b, c, d, f	e, r
≥ 2	the area above f will be expanded for the cop and		
	the area below e will be expanded for the robber		

Hence, the cop cover will include 10 and the robber cover 11 nodes. When the cop moves to d instead the sequence of computation is

time		cop cover	robber cover
0	d is expanded by the cop	d	$\emptyset$
0	r is expanded by the robber	d	r
1	r is not expanded by the cop since already covered		
	b, c, e are expanded by the cop	b, c, d, e	r
1	b, d, e are not expanded by the robber since already covered		
	a, f are expanded by the robber	b, c, d, e	a, f, r
≥ 2	the area below e will be expanded by the cop		
	the area above f will be expanded by the robber		

This results in a cop cover of 13 and a robber cover of 8. This means that the cop will move to d to gain a maximum heuristic value of 13. However, the only optimal move with respect to capture time is to move to b and the optimal value of the game is 6 cop moves (or 11 if both agents' moves are counted). If the cop does not choose b the robber can escape into the runoff above f. In contrast, when the cop does go to b the robber can only run into the larger area below e. Here, capture time is small.

The framework of Isaza *et al.*implements the cover computation slightly different. Here, a vertex is only declared as covered by an agent when he arrives at that vertex. Due to the fact that we

deal with the mathematical model here, all the edge costs are one (the objective is to minimize or maximize the number of turns). Furthermore, the evaluation does not always start with the cop. In contrast, the normal game is simulated, i.e. each agent moves when he is scheduled next. Now, consider Figure 5.23(b) with the cop at c, the robber at r and it is the cop's turn. Clearly, staying at c is dominated by either moving to a, b or d, in terms of optimality and covered area. Let the cop decide at time 0 to move to a. Then, we have the following computation sequence

time		cop cover	robber cover
0	r is expanded by the robber	$\emptyset$	r
1	a is expanded by the cop	a	r
1	k, l are expanded by the robber	a	k, l, r
2	b, c, e, g are expanded by the cop	a, b, c, e, g	k, l, r
2	g is not expanded by the robber since already covered		
	h, i are expanded by the robber	a, b, c, e, g	h, i, k, l, r
3	h, k are not expanded by the cop since already covered		
	d, f are expanded by the cop	a, b, c, d, e, f, g	h, i, k, l, r
	cop starts to expand the path above g		
≥ 3	robber expands the area below i		

Hence, the cop covers 10 vertices and the robber 9. Since the middle of the graph is symmetric we can immediately follow that when the cop moves to d in his first move the computation will calculate a cop cover of 11 and robber cover of 8. It remains to investigate the case when the cop moves to b. Then, we have

time		cop cover	robber cover
0	r is expanded by the robber	$\emptyset$	r
1	b is expanded by the cop	b	r
1	k, l are expanded by the robber	b	k, l, r
2	a, c, d, e, f are expanded by the cop	a, b, c, d, e, f	k, l, r
2	g, h, i are expanded by the robber	a, b, c, d, e, f	g, h, i, k, l, r
3	g, h, i are not expanded by the cop since already covered		
≥ 3	the robber expands the nodes above g and below i		

Therefore, when the cop moves to b the heuristic returns a cop cover of 6 and a robber cover of 13. Thus, the cop has to move to d to gain a maximum heuristic value of 11. However, the only optimal move to reduce capture time is to move to b and the optimal value of the game is 5 cop moves (or 9 if cop and robber moves are counted). When the cop moves to b the robber can neither escape in the areas above g nor below i. Any other move makes it possible for the robber to escape in either of the two areas which increases the number of moves necessary for capture.

It remains to remark that the subgraph above g and below i in the graph in Figure 5.23(b) could have been symmetric. Then, a Cover cop would have to break ties between moving to a or d whereas the optimal move remains b. We intentionally avoided the tie breaking by making these subgraphs asymmetric and therefore letting Cover choose one of the two moves deterministically.

Cover is also not an optimal algorithm for the robber even on 1-cop-win octile connected maps (cf. to the optimality of our algorithm, TrailMax, on such maps in Section 5.4). Using a map from the experiments in Subsection 5.5.1 that is 1-cop-win and has 1557 vertices, we simulated the same

speed game on 1000 randomly generated initial positions. Playing against an optimal cop at the same speed, both the original as well as our modified version only achieved a suboptimality of 23.65% and 98.07% on this map, respectively. This means that there is at least one strategy for the cop that achieves a smaller payoff against a Cover cop than the optimal value of the game. Hence, neither the original nor our modified Cover algorithm are optimal strategies for the robber on 1-cop-win octile connected maps.

5.6 Summary and open problems

Modern computer games require algorithms that compute move policies extremely fast to satisfy computation time requirements. Within this scope, computing optimal strategies is not a feasible approach. Hence, we have to investigate approximation algorithms. There are two major difficulties connected with the problem of computing such approximations. First, we would like to have a solution that resembles intelligent moves, hence is close to an optimal strategy. Second, the solution has to be computed quickly to satisfy computation time constraints.

Within this chapter, we have investigated several algorithms: Cover and our modification of Cover, hill climbing with perfect and maximum norm distance metric, Minimax, Dynamic Abstract Minimax and a variation, a random beacon algorithm and our new algorithms TrailMax and Dynamic Abstract TrailMax. We measured three quantities, practical performance in terms of node expansions and nodes touched, suboptimality and exploitability. The first two were evaluated in experiments on four different sets of maps running against a cop playing according to a Nash equilibrium. The latter was assessed by computing best responses on one set of maps and comparing them to the optimal solution.

Our analysis showed that our new approaches outperform all the previous techniques in quality, i.e. in suboptimality and exploitability. Furthermore, our new algorithms scale well with respect to computation time, meeting modern computer game computation time constraints. Hence, this work redefines the state-of-the-art in perfect information moving target search.

We discovered significant differences between our results and the reported behavior of the original Cover algorithm by Isaza *et al.* [36, 37]. We implemented multiple experiments to verify that the achieved capture times correspond to each other in both our and Isaza *et al.*'s frameworks. Conducting analogous suboptimality experiments for the one and two cop case showed similar performance to our results in Isaza's framework. These results suggest a low correlation of the original Cover heuristic to playing well in the game. Furthermore, we disproved Isaza's theorem stating that the Cover heuristic is optimal on 1-cop-win graphs.

There are several lines for future research. First, we only investigated algorithms for computation of target move policies. Most of the algorithms have either been already used for the cops or it seems likely that they can be altered to serve as pursuer algorithms. An evaluation of such algorithms with respect to our measures suboptimality and exploitability remains for future investigation. Note, that

it is unknown how a best response against a pursuer algorithm can be computed efficiently because the RA algorithm cannot be used in this case (cf. Section 3.1). Furthermore, having such algorithms at hand it would be of interest to perform a tournament evaluation, i.e. letting all pursuer algorithms play against all target algorithms.

Within our experiments, we have evaluated TrailMax's and DATrailMax's performance with respect to the theoretical measure of nodes expanded per turn. However, the algorithms do not actually spread their computation among the turns. Hence, future research is needed to either modify the computation of TrailMax or find a similar approximation whose computation can be distributed. Possible candidates are variations of TrailMax where the expansions around the robber's and cop's current vertex are limited to a certain radius or where points on the periphery of these areas are sampled.

Chapter 6

Mathematical cops and robbers

As far as the laws of mathematics refer to reality, they are not certain; and as far as they are certain, they do not refer to reality.

Albert Einstein (1879 - 1955)

Since the 1980's, the cops and robber problem has been given much attention in the mathematical literature. In Chapter 2 we only gave the necessary definitions for the previous chapters. Here, we will further engage in a discussion of the four major questions that are associated with the mathematical game of cops and robber and that have been addressed in the literature. The goal of this chapter is to outline more mathematical background and to present some results that have been by-products of the main work on this thesis.

The main problems connected to cops and robber are the following: First, we will consider the cop number of a graph (cf. Definition 2.4) in Section 6.1. We will outline known results for some classes of graphs and the order of the cop number with respect to the number of vertices in the graph. Second, the characterization of graphs where multiple cops can win is an open problem. We extend the idea from the one cop case (cf. Theorem 2.7) to the multiple cops case and give a few results on the characterization of some graphs where two cops have a winning strategy in Section 6.2. Third, we address the time to capture, called search time, and its worst case equivalent for given initial positions, called capture time (cf. Definition 2.5) in Section 6.3. Search time has been studied on graphs where one cop is sufficient to catch a robber and we review these results. Capture time is a new parameter and we establish tight bounds for the one cop case. Moreover, we computed the maximal search and capture time for graphs where two cops have a winning strategy with up to nine vertices. Fourth, we shortly discuss the complexity of computing the cop number and the complexity of computing strategies for the game. Finally, we outline some variations that have been addressed in the literature and link to references for the study of such modifications.

6.1 Cop number

Recall that the cop number of a graph G is the minimal number of cops that are required to catch a robber on that graph and is denoted by $c(G)$ (cf. Definition 2.4). This graph parameter has been studied extensively in the mathematical literature [1, 13, 14, 16, 17, 24, 25, 41, 50, 72]. However, we only review the most interesting results here for completion of the mathematical overview and to give more insight into the problem. The algorithms in previous chapters are used to compute strategies and are designed to work with a static number of cops, that is given as an input. We did not intend to design algorithms for a variable number of cops. However, the RA algorithm in Section 3.1 can also be used to determine the cop number of a graph and is, to the best of our knowledge, the computationally fastest algorithm to achieve this task.

The cop number is closely related to the treewidth $\text{tw}(G)$ of a graph. In fact, the treewidth is equal to the cop number in a variation of the game (cf. Section 6.5). Furthermore, the cop number in its original definition is known to be bounded by the treewidth.

Theorem 6.1 (by [41]) $c(G) \leq \text{tw}(G)/2 + 1$.

Treewidth is an important graph parameter since many known NP-hard problems are solvable in polynomial time for graphs with bounded treewidth [10]. We will also see in Section 6.4 that as a corollary of Theorem 6.1, a solution to the cops and robber problem is computable in polynomial time for graphs with bounded treewidth. However, it is NP-hard without this assumption.

The cop number is important with respect to network security. When modeling a network by a graph, one can ask the question of how secure this network might be when an intruder, e.g. virus, has to be found and eliminated quickly. Studies that are concerned with this measure of security are also performed on infinite or random graphs [13, 14, 50].

In general, it is possible to construct graphs with an arbitrarily high cop number. Aigner and Fromme [1] proved a first result with the help of graphs with high minimum degree. Their result was extended by Frankl [24] which we review here. Recall that the *girth* of a graph is the length of a shortest cycle contained in the graph.

Theorem 6.2 (by [24])
Let G have minimum degree d and girth at least $(8t-3)$ for some t. Then $c(G) > (d-1)^t$.

It is an interesting question if the cop number is bounded for certain types of graphs. Exact bounds are known for specific classes, e.g. Cayley graphs [24, 25, 34], cartesian products of trees [51] and for graphs where minors are excluded [6]. Furthermore, bridged graphs are 1-cop-win [7, 16], hence chordal graphs are 1-cop-win. An interesting result on classes of graphs that are not permitted to have a certain induced subgraph has been found recently by Joret *et al.* [41]. Recall that a *path* P_n is the graph $(\{1, \ldots, n\}, \{(i, i+1) | \ 1 \leq i \leq n-1\})$.

Theorem 6.3 (by [41])
The class of graphs that do not have an induced subgraph H has a bounded cop number if and only if every connected component of H is a path.

Bounding the length of an induced path means that every longer path through the graph must have a chord. Following from this theorem we know in particular that the classes with bounded length on induced paths have a bounded cop number. Notice however, that the inverse is not true. There are classes of graphs with arbitrary long induced paths that have a bounded cop number.

There are multiple graph parameters that have been used in trying to establish general bounds. Following Theorem 6.2, Andreae [5] proved that even with bounded minimum degree there are graphs with arbitrary high cop numbers, thus the minimum degree is not a graph parameter that can be used to produce a bound.

Theorem 6.4 (by [5])
For each $d, k \in \mathbb{N}$ with $d \geq 3$, there exists a d-regular connected graph G with $c(G) \geq k$.

However, there is a relationship between the (orientable) genus $g(G)$ and its cop number. The following result is especially interesting for planar graphs, i.e. graphs with genus 0. The resulting corollary has been proved earlier by Aigner and Fromme [1].

Theorem 6.5 (by [72]) $c(G) \leq \lfloor \frac{3}{2} g(G) \rfloor + 3$.

Corollary 6.6 (by [1]) $c(G) \leq 3$ *for any planar graph G.*

It is also interesting to investigate the order of the cop number when the graph increases in size. Meyniel conjectured that $c(G) = O(\sqrt{|V(G)|})$ (see Frankl [24]). However, the best currently proved result is established by Chiniforooshan [17].

Theorem 6.7 (by [17]) *The cop number of any n-vertex graph is in $O(\frac{n}{\log n})$.*

6.2 Characterization of cop-win graphs

The characterization of k-cop-win graphs has not yet been solved satisfactorily. The only known characterization is for 1-cop-win graphs and is based on recursive vertex removal [1, 64]. We have reviewed this characterization in Theorem 2.7. The question of how k-cop-win graphs can be characterized is open for $k \geq 2$ for the general problem as well as all variations (see Section 6.5). However, we will generalize the idea of vertex removal from the characterization of 1-cop-win graphs and engage in a discussion on potential characterizations of 2-cop-win graphs. We also give a construction of some 2-cop-win graphs. However, a complete characterization of 2-cop-win graphs remains an open problem. Knowing the characterization of 1-cop-win graphs, it is also interesting to investigate the number of 1-cop-win graphs with a given fixed number of vertices. This question has never

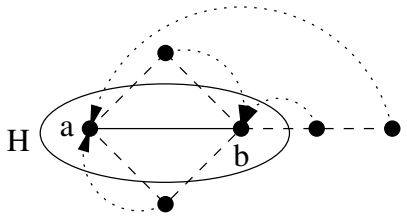


Figure 6.1: Example of a retract ϕ which is indicated by the dotted arrows.

been addressed. We compute the number of 1-cop-win graphs with up to nine vertices and therefore initiate the discussion on this problem.

The characterization of 1-cop-win graphs in Theorem 2.7 can be proved with the help of retracts which we will define below. Having this notation at hand it is possible to extend this theorem to k-cop-win graphs.

Definition 6.8 (homomorphism and retract)
A *homomorphism* is an edge preserving mapping from a graph G to a graph H. That means $\phi : V(G) \to V(H)$ is a homomorphism if

$$\forall (u, v) \in E(G) : (\phi(u), \phi(v)) \in E(H) \text{ or } \phi(u) = \phi(v).$$

For simplification ϕ will be denoted as $\phi : G \to H$. A graph G is said to be *homomorphic* to H if there exists a homomorphism from G to H.
A *retract* is a homomorphism from a graph G to an (induced) subgraph H of G that is the identity on H, i.e. $\forall v \in V(H) : \phi(v) = v$.

An example of a retract is depicted in Figure 6.1. The induced subgraph H is formed by a and b and the retract ϕ is indicated by dotted arrows. To check whether ϕ is indeed a retract we first have to check that it is a homomorphism. For every edge between two vertices in the original graph there is either an edge between the images of the vertices or the images are the same. Second, ϕ projects onto an induced subgraph of the original graph. Hence, ϕ is a retract.

Theorem 6.9 (by [8])
If G is connected and $\phi : G \to H$ is a retract, then

(a) $c(H) \leq c(G)$ *and*

(b) $c(G) \leq \max\{c(H), c(G - H) + 1\}$

The proof of (a) is simple. When playing on H the cops play on G and map all their actions into H by the given retract. The idea behind (b) is that $c(H)$ cops will catch the image of the robber under ϕ (in H). Then, one is left behind to stay on the image and $c(G - H)$ cops are send to search the remaining graph.

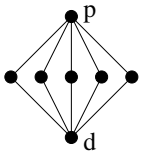


Figure 6.2: Example of a weak pitfall.

Let (p, d) be a pitfall in G (cf. Definition 2.6). Then we can construct a retract $\phi : G \to G - \{p\}$ such that $\phi(p) = d$ and the identity otherwise. Hence, pitfalls are retracts and therefore, by removing pitfalls from the graph, the cop number does not increase. Thus, Theorem 6.9(a) is a generalization of the forward implication of Theorem 2.7. However, the fact that adding a pitfall to a cop-win graph results in a cop-win graph cannot be seen from Theorem 6.9.

Another retract is a *weak pitfall*. Given a pair (p, d) of vertices, p is called a *weak pitfall* and d a *weak dominating vertex* if $N(p) \subseteq N(d)$. Notice that, in contrast to a pitfall, d and p can be at distance two from each other. A weak pitfall is depicted in Figure 6.2. It follows from Theorem 6.9 that adding or removing weak pitfalls to or from a 2-cop-win graph results in a 2-cop-win graph. But, even more is known. A *tandem* is a pair of cops that have to be at distance at most one after every (joint) move. Clarke [18] proved that the graphs that can be reduced to one vertex by removing pitfalls and weak pitfalls are tandem win.

Using general retracts gives rise to our construction of (some) 2-cop-win graphs that can easily be extended to the k-cop-win case.

Construction 6.10

Given a k-cop-win graph G, find a $(k-1)$-cop-win (not necessarily induced) subgraph H of G. Furthermore, find a connected, $(k-1)$-cop-win graph H' that is homomorphic to H, let ψ be the homomorphism. Obtain a new k-cop-win graph G' in the following way. Join G and H'. Given all the edges (h', v) where $v \in N[\psi(h')]$ and $h' \in H'$, add at least one of them to G'.

Proof. Construct a retract ϕ by setting $\phi(v) = v$ for all $v \in G$ and $\phi(h') = \psi(h')$ for all $h' \in H'$. It follows from Theorem 6.9 that G' is k-cop-win. □

An example of this construction is depicted in Figure 6.3. The dotted arrows signify the homomorphism ψ. The question of whether or not all 2-cop-win graphs can be constructed with this method remains open for further investigation.

Theorem 2.7 suggests another generalization to approach the characterization of k-cop-win graphs, in particular of 2-cop-win graphs. Define a *2-pitfall* to be a triple (p, d_1, d_2) such that $N[p] \subseteq N[d_1] \cup N[d_2]$. Note that d_1 does not necessarily have to be different from d_2. Since two cops can capture a robber in a 2-cop-win graph, the robber must be caught in a 2-pitfall at the end of the game. Thus, every 2-cop-win graph has at least one such 2-pitfall.

Unfortunately, assuming that removal of 2-pitfalls in any order yields a 2-cop-win graph is false.

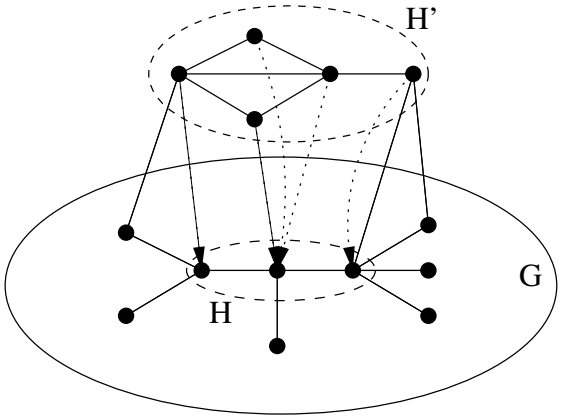


Figure 6.3: Visualization of Construction 6.10. ψ is indicated by the dotted arrows.

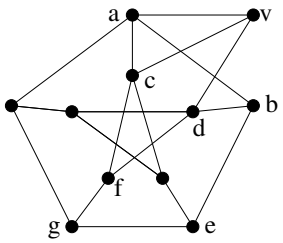


Figure 6.4: Petersen graph joint by an additional vertex v. This graph is 2-cop-win.

Consider the Petersen graph joint by an additional vertex v as depicted in Figure 6.4. Within this graph, v is a 2-pitfall with dominating vertices a and b. Notice that there are also other pairs of vertices that dominate v. When removing v from the graph, the result is the Petersen graph. The Petersen graph is the smallest 3-cop-win graph and it is known that all connected graphs with at most 10 vertices, except for the Petersen graph, are 2-cop-win [6]. This shows that removal of a 2-pitfall can increase the cop number. Moreover, removing a vertex from a cycle with four vertices C_4, which is 2-cop-win and every vertex is a 2-pitfall, yields a path with three vertices P_3, which is 1-cop-win. However, recall that a ℓ-cop-win graph is also a k-cop-win graph if $\ell \leq k$.

Conversely, adding a 2-pitfall to a 2-cop-win graph can increase and decrease the cop number as well. Consider the graphs depicted in Figure 6.5. The central vertex on the right is dominated by all pairs of vertices of the original C_4. However, the new graph is 1-cop-win since the cop can start in the central vertex and then catch the robber in the next move. Now, consider Figure 6.6. The graph without a is 2-cop-win, the cops start in c and b and can clear the graph from there. a is a 2-pitfall

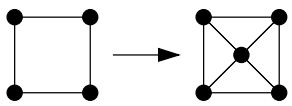


Figure 6.5: By adding a 2-pitfall to a 2-cop-win graph (square) it is possible to decrease the cop number.

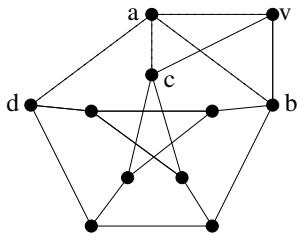


Figure 6.6: The full graph is 3-cop-win whereas the graph without a is 2-cop-win.

dominated by v and d. But, the full graph is 3-cop-win.

Notice that there are other 2-pitfalls in the graph in Figure 6.4 at b, dominated by v and e, and f, dominated by v and g. When removing b from the graph, the resulting graph has 10 vertices and is not equal to the Petersen graph and is thus 2-cop-win. The same holds true for f. Therefore, it is possible to conjecture the following.

Conjecture 6.11

For every 2-cop-win graph there is a 2-pitfall that can be removed from the graph without increasing the cop number.

If this holds true, conversely, it follows that for every 2-cop-win graph there is a construction of a sequence of 2-cop-win graphs via 2-pitfalls that yields the final graph. Such a construction for the graph in Figure 6.4 can be found in Figure 6.7. The added vertex in each step is encircled with a solid line whereas its two dominating vertices are marked with dotted lines.

Lemma 6.12

Let $\phi : G \to G - \{v\}$ be a retract. Then v is a 2-pitfall.
Let $\phi : G \to G - \{v, w\}$ be a retract. Then v and w are 2-pitfalls.

Proof. Due to the definition of a retract we have for all $u \in N(v)$

$$\phi(v) = u \quad \text{or} \quad (\phi(u), \phi(v)) = (u, \phi(v)) \in E(G - \{v\})$$

since ϕ is the identity on $G - \{v\}$. Therefore, it follows that $N(v) \subseteq N(\phi(v))$. Let $u \in N(v)$ then $N[v] = N(v) \cup \{v\} \subseteq N(\phi(v)) \cup N(u) \subseteq N[\phi(v)] \cup N[u]$. Hence, the first part follows.
Now, consider the second part of the theorem and let v and w not be neighbors in G. Analogously to the first part, it follows that v and w are 2-pitfalls. Let v and w be neighbors in G. Then, it follows that $N(v) - \{w\} \subseteq N(\phi(v))$ since ϕ is the identity on $G - \{v, w\}$. Therefore $N[v] = (N(v) - \{w\}) \cup \{v, w\} \subseteq N[\phi(v)] \cup N[w]$, thus $\phi(v)$ and w dominate v. Analogously, $\phi(w)$ and v dominate w. This proves the second part. □

Hence, for graphs where it is possible to remove some vertices v and w such that $\phi : G \to G - \{v\}$ or $\phi : G \to G - \{v, w\}$ are retracts, the conjecture is proved. This is due to the fact that v and w are 2-pitfalls, thus there are 2-pitfalls such that after their removal the resulting graph is

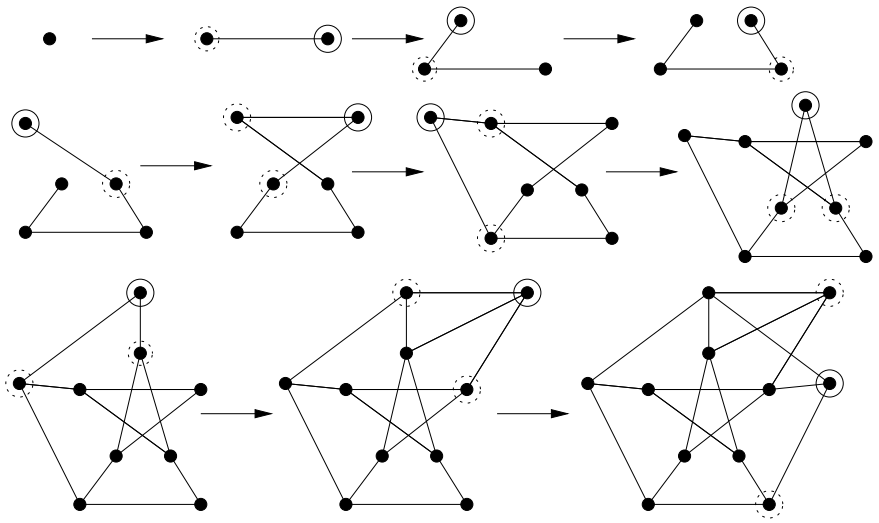


Figure 6.7: Sequence of 2-cop-win graphs that lead to a construction via 2-pitfalls. The new vertex in each step is circled with a solid line and its dominating vertices are circled with a dotted line.

number of vertices	2	3	4	5	6	7	8	9
connected graphs	1	2	6	21	112	853	11117	261080
(connected) (1)-cop-win graphs	1	2	5	16	68	403	3791	65561
connected planar graphs	1	2	6	20	99	646	5974	71885
(connected) planar (1)-cop-win graphs	1	2	5	15	59	294	1890	15304

Table 6.1: Number of connected, (1)-cop-win, connected planar and planar (1)-cop-win graphs with respect to the number of vertices.

2-cop-win. The proof or disproof of the conjecture for graphs with no such retracts remains open. An example graph where there are no retracts that abstract one or two vertices is C_7, the cycle with seven vertices. Here, all vertices are 2-pitfalls but non can be retracted.

Having a characterization for cop-win graphs at hand, it is interesting to investigate the number of cop-win graphs with respect to the number of vertices. Using the tool nauty [53] we generated the sets of all connected and connected planar graphs respective to their number of vertices and up to isomorphisms. Then, using an adjusted and optimized version of the RA algorithm from Section 3.1, we selected all the graphs that are 1-cop-win computationally. The results can be found in Table 6.1. The relation of the number of cop-win graphs due to the number of vertices has never been explored. Hence, our computation yields the first results on this question. Finding an explicit representation of these numbers is a very challenging problem that remains for future investigation.

6.3 Search and capture time

Recall that the search time of a graph G is the number of (joint) moves $c(G)$ cops need to catch a robber on G after selecting their initial positions and is denoted by $\mathrm{st}(G)$ (cf. Definition 2.5). The capture time $\mathrm{ct}(\mathbf{s})$ is the time the cops need to capture the robber given initial positions $\mathbf{s}$ and the

vertices	2	3	4	5	6	7	8	9
search time for 1-cop-win graphs	1	1	2	2	3	3	4	5
capture time for 1-cop-win graphs	1	2	3	4	5	6	7	9
search time for connected 2-cop-win graphs	0	1	1	1	2	2	2	3
capture time for connected 2-cop-win graphs	1	2	3	4	5	6	7	8

Table 6.2: Maximal search time for connected cop-win and 2-cop-win graphs due to the order of the graph.

capture time of a graph G, denoted by $\mathrm{ct}(G)$, is the maximum of $\mathrm{ct}(\mathbf{s})$ over all $\mathbf{s}$ in the state space (cf. Definition 2.5).

Bounding the search and capture time by the number of vertices in the graph is possible. Using the sets of all connected graphs we determined all connected cop-win and connected 2-cop-win graphs and their maximal search and capture time with the help of the RA algorithm in Section 3.1. The results can be found in Table 6.2. Knowing only the search time for cop-win graphs with at most five vertices, Bonato *et al.* [12] prove by induction that $\mathrm{st}(G) \leq |V(G)| - 3$. Using the computed results in Table 6.2 for cop-win graphs with at most seven vertices the same proof can be used to improve this bound.

Theorem 6.13 (first by [28])
Let G be cop-win and n be the number of vertices in G. If $n \leq 7$ then $\mathrm{st}(G) \leq \lfloor \frac{n}{2} \rfloor$. *If $n > 7$ then* $\mathrm{st}(G) \leq n - 4$.

Bonato *et al.* [12] give a simple construction of cop-win graphs with this maximal capture time. The graphs generated by this construction are planar, hence the theorem also holds for planar graphs. Gavenčiak [27, 28] proved the obtained search time values in Table 6.2 for at most seven vertices analytically. Furthermore, he characterizes cop-win graphs with maximal search time and studies the size of the set of all these graphs.

Capture time is a new parameter that has not been studied in the mathematical literature before. We give best possible bounds on the capture time for 1-cop-win graphs and generalize these bounds to some k-cop-win graphs in the following.

Theorem 6.14
Let G be cop-win and n be the number of vertices in G. If $n \leq 8$ then $\mathrm{ct}(G) \leq n - 1$. *If $n > 8$ then* $\mathrm{ct}(G) \leq 2n - 9$.

Proof. Let $n \leq 8$. The values have been proved computationally with the use of the RA algorithm in Section 3.1. An example graph with the worst case capture time is a P_n and spawning the robber and the cop on opposite sides of the path.
Let $n > 8$. The proof is by induction with the computed value of 9 as a base case for $n = 9$. Let G have $n + 1$ vertices and let (p, d) be a pitfall with its dominating vertex in G. Further, let $G' = G - \{p\}$ be the graph without p. Due to the characterization of cop-win graphs, G' is cop-win.

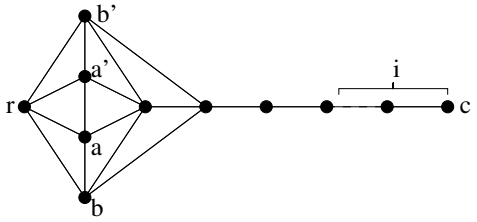


Figure 6.8: The construction of graphs M_i with maximal capture time.

We describe a winning strategy for the cop. He plays his strategy on G' whenever the robber is on G'. When the robber enters p the cop acts as if the robber would be at d and keeps playing his strategy on G'. Notice that every adjacent vertex of p is also adjacent to d, hence the simulation of the robber going to d instead of p is a valid move in G'. Assume capture has to occur in p. Then the cop captures the imaginary position of the robber in d first. If the robber is not at d he is at p. Then, the cop can capture the robber in one more move. When capture does not occur in p, then capture occurs in G'. Hence, when trying to catch the robber, the solution length can only increase by at most one when adding a new vertex.
Since the cop can be spawned anywhere in the graph, he can either be spawned in G' or on p. In the first case, he can start capturing the robber immediately with the above strategy. In the second case, he needs one additional move to get to G' and then executes the above strategy. Hence, the capture time increases by at most one because of the strategy and at most one because of the access to G'. Therefore, $\text{ct}(G) \leq \text{ct}(G') + 2 \leq 2n - 9 + 2 = 2(n+1) - 9$. □

This bound is the best possible. Trivially, for $n \leq 8$ using $G = P_n$ and spawning cop and robber on opposite sides of the path shows the tightness of the bound. For $n > 8$ consider the family of graphs M_i that can be constructed as depicted in Figure 6.8. Placing the robber at r and the cop at c results in maximal capture time. The cop runs towards the robber until he arrives at either a or a'. The robber then chooses to move to b' or b and the cop goes to a' or a, respectively. The chase then continues down the runoff until the robber gets caught in the original starting position of the cop.

Notice that the described strategy in the proof is nothing else than using a retract from $\phi : G \to G - \{p\}$, capturing the image of the robber under ϕ first and then moving onto p when the robber is not on $\phi(p)$. However, due to the nature of a pitfall, this technique cannot be directly applied to k-cop-win graphs. In the following, we will give an extended bound on the search and capture time when a retract is available.

Lemma 6.15

Let G' be a k-cop-win graph and $\phi : G' \to G$ be a retract such that $H = G' - G$ is at most $(k-1)$-cop-win. Then,

$$\begin{aligned} \text{st}(G') &\leq \text{st}(G) + \text{ct}(H) + \text{diam}(H) + 3 \\ \text{ct}(G') &\leq \text{ct}(G) + \text{ct}(H) + 2\,\text{diam}(H) + 5. \end{aligned}$$

Proof. As an example, consider the visualization in Figure 6.3 where the G' here is the graph G together with H' and we remove the part H' which is named H here. Furthermore, note that if H is disconnected we have $\text{diam}(H) = \infty$ and the lemma holds immediately. Thus, we assume H to be connected in the following.

Consider the case where the cops can choose their initial locations. They do so according to their strategy on G. Let r always denote the current position of the robber at the respective time. The cops first capture $\phi(r)$, i.e. the image of the robber's position under ϕ. This takes at most $\text{st}(G)$ moves. The capturing cop will stay on $\phi(r)$ for the rest of the game, guarding the image of the robber. When capture of $\phi(r)$ occurs, all other cops can ensure to be at most at distance one of this position. Hence, they can enter $\phi(H)$ by at most one more (joint) move. They then traverse $\phi(H)$ in the direction of a vertex $h' = \phi(h)$ that has a neighbor connected to h. Traversal of $\phi(H)$ takes at most $\text{diam}(\phi(H)) \leq \text{diam}(H)$ moves and entering H then takes at most two more moves. Afterwards, the robber is being chased down in H which will take at most $\text{ct}(H)$ moves. The first part of the theorem now follows.

Now, consider the case where the cops and the robber are spawned in any positions of the graph. If the cops are spawned in G, they immediately proceed with the above strategy. The only difference is that capture of $\phi(r)$ takes at most $\text{ct}(G)$ moves. If the cops are spawned in H, they traverse H to get to G. This takes at most $\text{diam}(H) + 2$ moves. They then proceed with the capture strategy outlined above. This proves the second part of the lemma and completes the proof. □

Corollary 6.16

Let G be a 2-cop-win graph that can be iteratively constructed via a sequence of 2-cop-win graphs $G_0, G_1, \ldots, G_k = G$ by Construction 6.10. Then,

$$\begin{aligned} \text{st}(G_i) &\leq 3(|V(G_i)| - |V(G_0)|) + \text{st}(G_0) \\ \text{ct}(G_i) &\leq 5(|V(G_i)| - |V(G_0)|) + \text{ct}(G_0). \end{aligned}$$

Proof. The proof is by induction on the iterative construction of G. Clearly, the claim holds for G_0. Assume the claim holds for i. Then let $H = G_{i+1} - G_i$ be the copy of the cop-win subgraph of G_i. Due to the construction H is cop-win and there exists a retract from G_{i+1} to G_i. Therefore, following from Lemma 6.15, we have

$$\text{st}(G_{i+1}) \leq \text{st}(G_i) + \text{ct}(H) + \text{diam}(H) + 3.$$

It is clear that $\text{diam}(H) \leq |V(H)| - 1$. Note that the construction prescribes that H is connected. Now, let $|V(H)| \leq 8$. Then it follows from Theorem 6.14 that $\text{ct}(H) \leq |V(H)| - 1$. Hence, by adding $|V(H)| \leq 8$ vertices to the graph the search time increases by at most

$$\text{increase}(|V(H)|) = |V(H)| - 1 + |V(H)| - 1 + 3 = 2\,|V(H)| + 1.$$

Now, let $|V(H)| > 8$. Then it follows from Theorem 6.14 that $\text{ct}(H) \leq 2|V(H)| - 9$. Hence, by adding $|V(H)| > 8$ vertices, the search time increases by at most

$$\text{increase}(|V(H)|) = |V(H)| - 1 + 2|V(H)| - 9 + 3 = 3\,|V(H)| - 7.$$

By calculating the values of the increase for $m \leq 8$ it is easy to see, that $\text{increase}(m) \leq 3m$ ($m \in N, m \geq 1$). Therefore, it follows that

$$\begin{aligned} \text{st}(G_{i+1}) &\leq \text{st}(G_i) + \text{increase}(|V(H)|) \\ &\leq \text{st}(G_i) + 3|V(H)| \\ &\leq 3(|V(G_i)| - |V(G_0)|) + \text{st}(G_0) + 3(|V(G_{i+1})| - |V(G_i)|). \end{aligned}$$

The first part now follows. The proof of the second part is analog. □

In particular, we can bound the search and capture time for 2-cop-win graphs that can be constructed by a sequence starting in $G_0 = P_2$. Then, we have

$$\text{st}(G) \leq 3|V(G)| - 6 \quad \text{and} \quad \text{ct}(G) \leq 5|V(G)| - 9$$

since $\text{st}(P_2) = 0$ and $\text{ct}(P_2) = 1$ when two cops are used. Unfortunately it is not easy to extend this result recursively to k-cop-win graphs that can be constructed with Construction 6.10. This is due to the fact that the proof depends on the search time of H, that is unknown for graphs that do not obey the construction. Even though we know that G can be constructed it is unclear whether H can be.

It is interesting to ask whether there are graph properties other than the number of vertices that can be used to establish bounds on the search and capture time. A first parameter that comes into mind is the diameter.

Theorem 6.17 (by [32])
For every $s \geq 1$ there is a finite, diameter two, chordal, hence cop-win, graph G_s on which the robber can survive for at least s moves, i.e. $\text{st}(G_s) \geq s$.

Since $\text{ct}(G) \geq \text{st}(G)$ this also proves that the diameter is not useful for either search or capture time. Moreover, the length of the longest path or the longest cordless path are not related to search or capture time. With regards to the longest path, consider the complete graph K_n for which $\text{st}(K_n) = \text{ct}(K_n) = 1$ but the longest path is of size n. With regards to the longest cordless path, consider the graph obtained by a path P_n joined with a vertex that is connected to all path vertices for which the search time is one and capture time is two but the longest cordless path is of length n.

The TrailMax function from Section 5.4 approximates the capture time for given initial positions. We have shown that this function computes the exact capture time for cop-win octile maps (cf. Theorem 5.2), hence also the search time given the optimal initial positions. However, we have also given examples of general cop-win graphs where TrailMax underestimates arbitrarily.

6.4 Complexity

With regards to complexity, there are two major problems connected with the game of cops and robber. First, we will investigate the complexity of determining the cop number of a graph. Second, the complexity of determining the search and capture time of a graph given a fixed number of cops will be of interest.

The calculation of the search and capture time is possible via the construction of W's (Chapter 2). The problem is fixed-parameter tractable with the fixed parameter being the number of cops, i.e. its runtime is in $O(n^{O(k)})$ where n is the order of the graph and k is the number of cops. A proof can be found in the paper of Berarducci and Intrigila [8]. A practical formulation of this algorithm has been given by Hahn and McGillivray [33] which we used as a baseline algorithm in Chapter 3.

The complexity of determining the cop number of a graph has not been completely solved. Goldstein and Reingold [30] proved the following

Theorem 6.18 (by [30])
The problem of determining if $c(G) \leq k$ on an undirected graph G with given initial positions is EXPTIME-complete.

This is particularly interesting because we used given initial positions in the previous chapters. Hence, we cannot hope to determine the cop number in feasible time and then compute strategies for them. This justifies our approach that we assume that a sufficient number of cops is given as an input parameter to the algorithms in previous chapters.

The complexity of the game without given initial positions was only solved recently by Fomin *et al.* [22].

Theorem 6.19 (by [22])
The problem of determining if $c(G) \leq k$ on an undirected graph G without given initial positions is NP-hard.

Goldstein's conjecture that the full mathematical game, i.e. with initial position selection, might be EXPTIME-complete remains an open problem. Nevertheless, when the game is played on directed graphs, Goldstein and Reingold were able to prove that the full game is EXPTIME-complete [30].

6.5 Variations

There are many variations studied in the literature and a complete discussion is beyond the scope of this thesis. For more detailed reviews see [4, 23, 31]. Nevertheless, a list of the most common possible deviations from the original game definition will be presented below and will be linked to references that are mainly concerned about these games. We include these variations to extend the focus from the very restrictive mathematical definition of the cops and robber game to more general

pursuit games. In fact, we have already used a game where the cops move at faster speed than the robber for our experiments in Chapter 5.

Continuity. The pursuit evasion game can be played in a continuous space with continuous actions. Given movement models the game becomes a game of differential equations, hence the name differential games. Isaacs [35] is the standard book for this type of game. Furthermore, there are multiple studies of games with mixed discrete and continous assumptions (cf. [3, 9] and references therein).

Graph. The original version is played on an undirected graph. There are several results for the case of directed graphs. Besides the EXPTIME-completeness of the full game [30] there is nearly nothing known on characterizations of k-cop-win graphs even for $k \geq 1$. To obtain measures of security in networks the cop number has been studied on random graphs [14, 50]. Furthermore, infinite graphs have been given attention in [13, 32].

Information. In the original definition, the cops and robber have full information during all stages of the game. This can be modified in several ways. Studied possibilities are decreasing the visibility of cops and/or robber [40], witnessed versions where witnesses/sensors provide information to the cops [19] similar to a Scotland Yard game, and no information about the robber's position at all [54, 65].

Movement. Cops and robber might not be constrained to move along edges or at the same speed. Seymour and Thomas [73] study a version where the cops use helicopters to go from one vertex to the next, hence are not bound on the graph at all, and the robber can run at infinite speed at any time. They establish the connection between cop number and treewidth.

Theorem 6.20 (by [73])
G has treewidth at least $(k-1)$ if and only if less than k cops cannot catch a robber in the Seymour and Thomas game.

Another common model is that the robber can move at any speed at any time, but the cops are constrained to the graph. This can also be thought of as cops having to clean a graph from a toxic gas. The class of these games is also known as sweeping graph games and has received much attention within the literature [54, 65]. Furthermore, the cops and/or robber might have different speed [22, 63]. The above mentioned complexity results for undirected graphs also hold for a faster robber and several other classes of graphs [22]. Clarke and Nowakowski studied a series of cops and robber games with many different sorts of constraints on movement [18].

In the model studied in the previous chapters, the cops and robber move alternatingly. This can be changed to a simultaneous action game. We will study the classical cops and robber game, made simultaneous, in Chapter 7. Another source of simultaneous games are sweeping games. Furthermore, the discretizations of continuous-time continuous-action versions are played simultaneously. Here, the cops and the robber move simultaneous in continuous motion. When no analytical solution

to the describing differential equations can be found, the problem is solved with numerical methods. Thus, space is discretized but the action sets remain infinite (usually some sort of compactness is assumed). The game is therefore played on some type of grid but with infinitely many actions. Then, transition probabilities that are induced by the underlying differential equations are included, i.e. the probability that under an action the state is transferred into another state. It has been shown that there exist equilibria in pure strategies for both players and how they can be computed [47, 66]. A pure strategy in this game can be interpreted as a function that maps the position of an agent to the agent's action which is a discrete solution to the implied differential equations.

Objective. In the original cops and robber game, the robber tries to survive as long as possible. There are other variations where the robber is inert [75] or the cops want to guard a subgraph and the robber tries to enter this region [21]. Furthermore, Bonato and Chiniforooshan [11] generalize the bound on the cop number given in Theorem 6.7 to a game where the cop can shoot the robber from a fixed distance, i.e. the game ends as soon as the cops get within a certain distance of the robber.

6.6 Summary and open problems

The cops and robber game has been studied in many variations within the mathematical literature. We reviewed the classical and most common definition. There are four major problems connected with this game. First, the number of cops required to catch a robber. We have reviewed known results and given an overview of the ongoing research. It is very difficult to achieve tight bounds on the number of cops required to catch a robber with respect to some graph properties.

Second, we investigated the characterization of k-cop-win graphs. We explained the known characterization for $k = 1$ and gave two methods for generalization of these concepts, namely via retracts and k-pitfalls. We characterized some 2-cop-win graphs by using these generalizations. However, the general problem, that has been unsolved for more than two decades, of how k-cop-win graphs can be characterized, remains open.

Third, we studied the search time of a graph and the capture time, a new parameter. The search time is known for 1-cop-win graphs. We established tight bounds on the capture time for 1-cop-win graphs. Although we give a tool to bound the capture and search time for some k-cop-win graphs and deduce some bounds for $k = 2$, the problem of establishing bounds for $k \geq 2$ is not yet satisfactorily solved.

Fourth, we reviewed the complexity of computing the cop number of a graph and outlined the complexity for computation of the search time and capture time for each state. Although we know that the entire game, i.e. with initial position selection, is NP-hard it is not known if the game is EXPTIME-complete. However, given initial positions, the game is EXPTIME-complete.

Finally, we discussed several different variations of the game and pointed to references that span a broader field of research connected to the cops and robber game in particular and pursuit games in general.

Chapter 7

Simultaneous cops and robber

Problems worthy of attack prove their worth by fighting back.

Paul Erdös (1913 - 1996)

In this chapter, we will investigate a variation of the original cops and robber game where both players move simultaneously. Our goal is to compute optimal solutions, i.e. a Nash equilibrium. It is well known that a Nash equilibrium might not exist in pure but only in mixed strategies. Since the solution to the alternating game is a pure strategy, the RA algorithm from Section 3.1 cannot be used to approach optimal solutions for the simultaneous game. In the following, we will first define the term strategy for this game variation. Second, we will discuss how a general linear program (LP) can be deduced and why it cannot be solved in practice. Third, we will review Littman's Markov Game formulation [48] and outline how the cops and robber problem can be formulated as an undiscounted case. Value iteration is used to solve the problem but since our formulation is undiscounted there is no standard proof of correctness. We introduce an equivalent formulation of finiteness of the expected value of the simultaneous game and prove that value iteration converges to the correct game values under this assumption.

Since both players move simultaneously we can disregard the tag for whose turn it is in the state space and have to introduce the notion of actions. Note that in the alternating game, an action is the same as a pair of subsequent states since the players move sequentially. However, in the simultaneous game, two actions are taken simultaneously. Hence, the state trajectory is defined by tuples of actions but an action for only one player has to be defined differently than before. Let $\mathbf{s} \in S$ be a state (without the turn tag). Then we denote the different possibilities to move for the robber in $\mathbf{s}$ as *actions* $A_{\mathbf{s}}$ and for the cops as actions $O_{\mathbf{s}}$. Let A be the distinct union of all action sets for the robber for all states and analogously O for the cops.

Definition 7.1 (strategy for the simultaneous move game)
Let G be the graph that the game is being played on, S its state space, A the action set for the robber, O the action set for the cops and $\mathcal{P}(\cdot)$ be the space of probability distributions over a given set. Then, a *stationary strategy* for the robber player (the cop player) is a function $\theta : S \to \mathcal{P}(A)$

$(\theta : S \to \mathcal{P}(O))$ such that

$$\forall \mathbf{s} \in S : \sum_{a \in A_{\mathbf{s}}(O_{\mathbf{s}})} \theta(\mathbf{s})_a = 1. \tag{7.1}$$

Note that a strategy in the Nash equilibrium sense is a strategy that keeps track of the history of the game (hence a strategy on the trajectories of the game). Here, both players have full information on the state of the game but no knowledge of the opponent's current action. Hence, optimal play consists of optimal moves in each step given the expected value of the subsequent playout. Therefore, an optimal strategy due to a Nash equilibrium can be represented as a one-step strategy, i.e. a stationary strategy. Since our goal is to compute a Nash equilibrium in optimal strategies and these are equivalent to optimal stationary strategies we will use the term *strategy* instead of stationary strategy in the following.

Mixed strategies indicate the probability with which an agent, being in a state $\mathbf{s}$, should take an action a. Hence, we can interpret this as $\theta' : S \times A \to [0, 1]$ with $\theta'(\mathbf{s}, a) = \theta(\mathbf{s})_a$ for the robber and analogously for the cops. To simplify the presentation we will use this more intuitive notation rather than the above definition in the following.

If the robber and a cop are adjacent to each other it is possible that both agents declare to move along their joining edge in opposite directions during the simulation of the game. In this case the game ends and the robber is declared as being captured.

The immediate approach to finding optimal strategies is to generate a linear program (LP) that solves the game. A (stationary) *pure strategy* for the robber (the cops) is a strategy such that

$$\forall \mathbf{s} \in S \quad \exists a \in A_{\mathbf{s}}(O_{\mathbf{s}}) : \quad \theta(\mathbf{s}, a) = 1.$$

Since there can only be one such action a for which $\theta(\mathbf{s}, a) = 1$ due to (7.1), a pure strategy is a mapping from a state to an action to be taken in the state. This corresponds to the definition of a strategy in the alternating game (see Definition 3.1).

In theory, it is possible to generate all pure strategies $\theta_{C,1}, \ldots, \theta_{C,p}$ for the cops and $\theta_{R,1}, \ldots, \theta_{R,q}$ for the robber. Then, playing these strategies against each other and determining the payoff yields the matrix game

$$B = \begin{pmatrix} \text{payoff}(\theta_{C,1}, \theta_{R,1}) & \ldots & \text{payoff}(\theta_{C,1}, \theta_{R,q}) \\ \vdots & \ddots & \vdots \\ \text{payoff}(\theta_{C,p}, \theta_{R,1}) & \ldots & \text{payoff}(\theta_{C,p}, \theta_{R,q}) \end{pmatrix}.$$

Note that some of the values in $\mathbf{B}$ can be infinite. We want to find the solutions to

$$\min_{\mathbf{x} \in \mathbb{R}^p} \max_{\mathbf{y} \in \mathbb{R}^q} \mathbf{x}^\mathsf{T} \mathbf{B} \mathbf{y} \quad \text{and} \quad \max_{\mathbf{y} \in \mathbb{R}^q} \min_{\mathbf{x} \in \mathbb{R}^p} \mathbf{x}^\mathsf{T} \mathbf{B} \mathbf{y} \quad \text{with} \tag{7.2}$$
$$x, y \geq 0, \quad \sum_{i=1}^{p} x_i = 1, \quad \sum_{j=1}^{q} y_j = 1.$$

These solutions to (7.2) indeed induce mixed strategies. Having a solution $\mathbf{x}$ to the left side of (7.2) we can formulate a mixed strategy θ_C for the cops such that

$$\theta_C(\mathbf{s}, a) = \sum_{i=1}^{p} x_i \, \theta_{C,i}(\mathbf{s}, a)$$

and analogously θ_R for the robber due to a solution $\mathbf{y}$ to the right side of (7.2)

$$\theta_R(\mathbf{s}, o) = \sum_{j=1}^{q} y_j \, \theta_{R,j}(\mathbf{s}, o).$$

Since the matrix game $\mathbf{B}$ has a finite number of actions, a Nash equilibrium exists for the matrix game [59, 62]. That means the two terms in (7.2) are equal (to the value of the game). Clearly, the value of the game and the induced (stationary) strategies form an equilibrium in the original game. The left side of (7.2) can be reformulated into a LP to find a strategy for the cops

$$\begin{aligned} \min_{t\in\mathbb{R},\mathbf{x}\in\mathbb{R}^p} \quad & t + \mathbf{0}^\mathsf{T}\mathbf{x} && \text{such that} \\ \forall 1 \le j \le q: \quad & t \ge \mathbf{x}^\mathsf{T}\mathbf{B}_{\cdot\mathbf{j}}, \\ \forall 1 \le i \le p: \quad & x_i \ge 0, \\ & \textstyle\sum_{i=1}^{p} x_i = 1, \end{aligned} \tag{7.3}$$

where $\mathbf{B}_{\cdot\mathbf{j}}$ denotes the jth column of $\mathbf{B}$. Analogously we can formulate a LP to find an optimal strategy for the robber

$$\begin{aligned} \max_{s\in\mathbb{R},\mathbf{y}\in\mathbb{R}^q} \quad & s + \mathbf{0}^\mathsf{T}\mathbf{y} && \text{such that} \\ \forall 1 \le i \le p: \quad & s \le \mathbf{B}_{\mathbf{i}\cdot}\mathbf{y}, \\ \forall 1 \le j \le q: \quad & y_j \ge 0, \\ & \textstyle\sum_{j=1}^{q} y_j = 1, \end{aligned} \tag{7.4}$$

where $\mathbf{B}_{\mathbf{i}\cdot}$ denotes the ith row of $\mathbf{B}$.

The problem is that there are exponentially many pure strategies in the number of vertices of the graph, hence $\mathbf{B}$ cannot be generated efficiently and is too large. This renders the approach of solving a directly formulated LP like (7.3) and (7.4) infeasible.

7.1 Markov Game formulation

A different approach for theses type of games has been formulated by Littman [48]. We will review his formulation of a *Markov Game* in the following and outline how this can be used to solve our game.

Let $R(s, a, o)$ be the reward or immediate payoff for one of the two players (in our case the robber) when being in state $\mathbf{s} \in S$ and the robber chooses action $a \in A$ and the cops choose $o \in O$. The reward when the robber takes action a and the cops choose (joint) action o is the maximum of the length of the edges that the agents take to get to their next locations (see the discussion on time and payoff at the beginning of Chapter 3). If all agents decide to stay, the reward is set to one. Hence, the rewards are always greater than 0. Let $GV(\mathbf{s})$ be the expected reward for the robber in state $\mathbf{s}$, i.e. the game value when the game starts in state $\mathbf{s}$ (cf. Definition 3.3). Then we can formulate an

equilibrium, the Bellman equations, as follows

$$GV(\mathbf{s}) = \min_{\pi \in \mathcal{P}(O_\mathbf{s})} \max_{a \in A_\mathbf{s}} \sum_{o \in O_\mathbf{s}} \pi_o \, Q(\mathbf{s}, a, o) \tag{7.5a}$$

$$Q(\mathbf{s}, a, o) = R(\mathbf{s}, a, o) + \gamma \, GV(\mathbf{s}') \tag{7.5b}$$

where $\mathbf{s}'$ is the state that is reached when taking actions a and o from state $\mathbf{s}$ and $0 \leq \gamma \leq 1$ is a discount factor. In fact, this equilibrium defines a γ-discounted game. When $\gamma = 1$, it is the usual cops and robber game.

Note that this equilibrium corresponds to the left term of (7.2). A formulation corresponding to the right term of (7.2) can be obtained by changing the roles of the $\min$ and $\max$ in (7.5a) with the stochastic choice for the maximization. Note further that Littman formulated the Bellman equations as extensions for general Markov Decision Processes (MDPs), thus including probabilistic transitions from one state to another in (7.5b). However, since the environment we consider here is completely deterministic, we do not need this extra complexity.

Given the values of Q the equation (7.5a) can be solved using linear programming approaches. We used the algorithm due to Robinson within our implementation [70]. The entire system (7.5) can be solved by the following algorithm.

Algorithm 7.1 (value iteration)
Start with an initial estimate GV_0 for GV and subsequently generate new estimates by treating the equality signs in (7.5a) and (7.5b) as assignment operators, i.e.

$$GV_{i+1}(\mathbf{s}) = \min_{\pi \in \mathcal{P}(O_\mathbf{s})} \max_{a \in A_\mathbf{s}} \sum_{o \in O_\mathbf{s}} \pi_o \, (R(\mathbf{s}, a, o) + \gamma \, GV_i(\mathbf{s}')). \tag{7.6}$$

Note that this algorithm is a fixpoint iteration but we prefer the name value iteration which is widely used terminology in computing science.

Let GV_0 be a first estimate and GV_i be the estimates of subsequent iterations. Then it has been shown for $0 \leq \gamma < 1$ that the sequence $(GV_i)_{i \in \mathbb{N}}$ converges linearly [49, 74]. More precisely, there exists a unique fixpoint GV^* and

$$\|GV^* - GV_i\| \leq \gamma^i \|GV^* - GV_0\|,$$

where $\| \cdot \|$ is the maximum norm [78].

A *myopic* policy induced by GV is a pair of strategies that obtain game values of GV for the γ-discounted game. Hence, they are obtained by a one move lookahead with evaluations of subsequent states by GV. Let GV^π be the evaluation of policy π, i.e. GV^π is the solution to system (7.5) when $\min$ and $\max$ are substituted by play according to the strategies in π. Informally, GV^π is the playout of the strategies in π in the γ-discounted game. Now, let π_i be the myopic policy to GV_i. Then, if $0 \leq \gamma < 1$, it is possible to bound the error between the strategies in π_i and optimal play (in the γ-discounted game) by the error of GV_i in subsequent iterations [78]

$$\|GV^{\pi_i} - GV^*\| \leq \frac{2\gamma}{1-\gamma} \|GV_{i+1} - GV_i\|.$$

Therefore, when the difference in two subsequent iterations of the value iteration is small we also know that the difference in performance of a myopic policy, with respect to the current GV estimate, and optimal play is small.

Here, we are interested in the undiscounted game, hence $\gamma = 1$. The above convergence results and error bounds can only be obtained for $0 \leq \gamma < 1$. Therefore, we will prove that if there is a strategy for the cops to win the simultaneous game the value iteration converges to a fixpoint and this fixpoint defines a Nash equilibrium.

Definition 7.2

If there is a (stationary) strategy θ_C for the cops and a number $t \in \mathbb{N}$, such that for every (stationary) strategy θ_R for the robber the game (starting in any initial position $\mathbf{s}_0 \in S$) ends after t steps with probability $p > 0$, we say the cops can *win* the game.

Lemma 7.3

The expected value of the game is finite if and only if the cops can win the game (as in Definition 7.2).

Proof. We first prove that when the cops can win the expected value is finite. Let

$$\|R\| = \max_{\mathbf{s} \in S, a \in A_{\mathbf{s}}, o \in O_{\mathbf{s}}} R(\mathbf{s}, a, o).$$

Then, the maximum reward after t steps is $t\|R\|$. Since the cops can end the game after at most t steps with probability $p > 0$ we have

$$\begin{aligned} \mathbb{E}(GV(\mathbf{s})) &\leq t\|R\| + (1-p)(t\|R\| + (1-p)(\ldots)) \\ &= t\|R\| \left(1 + (1-p) + (1-p)^2 + \ldots\right) \\ &= \frac{1}{p} t\|R\|. \end{aligned}$$

Thus, the expected value of the game is finite.

Now, let the expected value of the game be finite. Assume there is no strategy for the cops that fulfills Definition 7.2. Then, for all strategies θ_C for the cops, there is a strategy θ_R such that there is no t for which the game ends after t steps with nonzero probability. Since the state space S is finite, we have that there has to be at least one initial position $\mathbf{s}_0$ for which the game does not end for any number of steps (otherwise, we would have a $t_{\mathbf{s}}$ for every $\mathbf{s} \in S$ and hence $t = \max_{\mathbf{s} \in S} t_{\mathbf{s}}$). Therefore, we have $\mathbb{E}(GV(\mathbf{s}_0)) = \infty$ which is a contradiction to our assumption. Thus, the cops can win the game. □

When using k cops on a graph that is not k-cop-win, there is a strategy for the robber to evade capture forever in the alternating game that can be extended to the simultaneous game. The robber plays according to his strategy in the alternating game. He assumes the cops stay still in their first move and declares his answer as his first move in the simultaneous game. He then interprets the actual simultaneous move of the cops as the next alternating move. It is easy to see that the robber can evade capture forever with this strategy.

Therefore, it is a necessary condition to have k cops when playing on a k-cop-win graph to be able to bound the expected value of the game. By exploiting the characterization of 1-cop-win graphs (cf. Theorem 2.7) it seems likely that we can prove that the expected value of the game for such graphs is finite. Unfortunately, there is no known characterization for k-cop-win graphs ($k \geq 2$). Therefore, it is a hard problem to answer whether by playing with k cops on a k-cop-win graph the expected value of the game becomes finite or not. This question remains for future research. The above definition provides a strong assumption that enables us to prove the convergence of value iteration in the following. It is however unclear, when this assumption holds.

It remains to remark that in the case where the expected value of the game is not bounded everywhere, we need to set $\gamma < 1$ to ensure finiteness of GV and Q in (7.5). However, the values in GV are rather uninteresting since large parts of the state space are assigned the same expected value (the maximal value that the value iteration converges to).

Definition 7.4

According to system (7.5) define the following operator $T : (S \to \mathbb{R}) \to (S \to \mathbb{R})$

$$T(F)(\mathbf{s}) = \min_{\pi \in \mathcal{P}(O_{\mathbf{s}})} \max_{a \in A_{\mathbf{s}}} \sum_{o \in O_{\mathbf{s}}} \pi_o(R(\mathbf{s}, a, o) + F(\mathbf{s}')),$$

where $\mathbf{s}'$ is the state reached from $\mathbf{s}$ via actions a and o.

Lemma 7.5

Let (θ_r, θ_c) be a Nash equilibrium and GV^* be the induced game values, i.e. $GV^*(\mathbf{s})$ is the game value when the play starts in $\mathbf{s}$. Then GV^* is a fixpoint of T.

Proof. The lemma follows by the fact that optimal strategies are optimal stationary strategies. Note that it does not matter if the game is played with sufficiently many cops because then some of the values in GV^* are just infinite. □

Lemma 7.6

Let $F', F : S \to \mathbb{R}$ such that $F' \geq F$. Then,

$$0 \leq T(F')(\mathbf{s}) - T(F)(\mathbf{s}) \leq \|F' - F\|,$$

where $\|\cdot\|$ is the maximum norm.

Proof. Without loss of generality let $\mathbf{s} \in S$ be fixed. In the following, $\|\cdot\|$ will denote the maximum norm and let $c = \|F' - F\|$. Due to T's definition $T(F)(\mathbf{s})$ is the solution value to a matrix game with entries from F. Therefore, the claim of the lemma means that the game values of the two matrix games can only differ by at most the maximum difference in the entries of the two matrices. Consider the matrix game $\mathbf{B} = (R(\mathbf{s}, a, o) + F(\mathbf{s}'))_{o \in O_{\mathbf{s}}, a \in A_{\mathbf{s}}}$ and let further $\mathbf{B}' =$

$(R(\mathbf{s},a,o)+F'(\mathbf{s}'))_{o\in O_\mathbf{s},a\in A_\mathbf{s}}$. Then, $0\le B'_{i,j}-B_{i,j}\le c$ and we can rewrite

$$T(F')(\mathbf{s}) = \min_{\mathbf{x}\in\mathbb{R}^p}\max_{\mathbf{y}\in\mathbb{R}^q}\mathbf{x}^\mathsf{T}\mathbf{B}'\mathbf{y} \tag{7.7a}$$

$$T(F)(\mathbf{s}) = \min_{\mathbf{x}\in\mathbb{R}^p}\max_{\mathbf{y}\in\mathbb{R}^q}\mathbf{x}^\mathsf{T}\mathbf{B}\mathbf{y}, \quad \text{such that} \tag{7.7b}$$

$$\sum_{i=1}^{p} x_i = 1,\ \mathbf{x}\ge 0, \qquad \sum_{j=1}^{q} y_i = 1,\ \mathbf{y}\ge 0, \tag{7.7c}$$

where $p=|O_\mathbf{s}|$ and $q=|A_\mathbf{s}|$. Let $\mathbf{x}\in\mathbb{R}^p$ and $\mathbf{y}\in\mathbb{R}^q$ satisfying the unity conditions (7.7c). Then we have

$$\begin{aligned}
\mathbf{x}^\mathsf{T}\mathbf{B}'\mathbf{y} &= \sum_{i=1}^{p}\sum_{j=1}^{q} x_i B'_{i,j} y_j \\
&\le \sum_i\sum_j x_i(B_{i,j}+c)y_j = \sum_i\sum_j x_i B_{i,j}y_j + \sum_i\sum_j x_i c y_j \\
&= \sum_i\sum_j x_i B_{i,j}y_j + c\sum_i x_i\sum_j y_j = \sum_i\sum_j x_i B_{i,j}y_j + c \\
&= \mathbf{x}^\mathsf{T}\mathbf{B}\mathbf{y} + c.
\end{aligned}$$

Since $\mathbf{x}$ and $\mathbf{y}$ were arbitrary we can follow that

$$\begin{aligned}
\max_{\mathbf{y}'\in\mathbb{R}^q}\mathbf{x}^\mathsf{T}\mathbf{B}'\mathbf{y}' &\le \max_{\mathbf{y}'\in\mathbb{R}^q}\mathbf{x}^\mathsf{T}\mathbf{B}\mathbf{y}' + c \\
T(F')(\mathbf{s}) = \min_{\mathbf{x}'\in\mathbb{R}^p}\max_{\mathbf{y}'\in\mathbb{R}^q}(\mathbf{x}')^\mathsf{T}\mathbf{B}'\mathbf{y}' &\le \min_{\mathbf{x}'\in\mathbb{R}^p}\max_{\mathbf{y}'\in\mathbb{R}^q}(\mathbf{x}')^\mathsf{T}\mathbf{B}\mathbf{y}' + c = T(F)(\mathbf{s}) + c,
\end{aligned}$$

where $\mathbf{x}'$ and $\mathbf{y}'$ obey (7.7c). We have $\mathbf{x}^\mathsf{T}\mathbf{B}\mathbf{y}\le\mathbf{x}^\mathsf{T}\mathbf{B}'\mathbf{y}$ since $0\le B'_{i,j}-B_{i,j}$. Hence,

$$\begin{aligned}
\max_{\mathbf{y}'\in\mathbb{R}^q}\mathbf{x}^\mathsf{T}\mathbf{B}\mathbf{y}' &\le \max_{\mathbf{y}'\in\mathbb{R}^q}\mathbf{x}^\mathsf{T}\mathbf{B}'\mathbf{y}' \\
T(F)(\mathbf{s}) = \min_{\mathbf{x}'\in\mathbb{R}^p}\max_{\mathbf{y}'\in\mathbb{R}^q}(\mathbf{x}')^\mathsf{T}\mathbf{B}\mathbf{y}' &\le \min_{\mathbf{x}'\in\mathbb{R}^p}\max_{\mathbf{y}'\in\mathbb{R}^q}(\mathbf{x}')^\mathsf{T}\mathbf{B}'\mathbf{y}' = T(F')(\mathbf{s}),
\end{aligned}$$

where $\mathbf{x}'$ and $\mathbf{y}'$ fulfill the unity conditions (7.7c). This completes the proof. □

Lemma 7.7

Let $GV^*<\infty$ be a fixpoint of T such that $GV^*(\mathbf{s})=0$ for all terminal states $\mathbf{s}$, i.e. states where there cannot be taken any actions or all actions lead back to $\mathbf{s}$. If the cops can win the game (due to Definition 7.2) then GV^* is equal to the equilibrium value of the game and the induced myopic strategies of GV^* are optimal.

Proof. Let GV be a fixpoint of T with the above requirements. Let further $\mathbf{s}_0\in S$ be the initial state. Recall, that

$$T(F)(\mathbf{s}) = \min_{\pi\in\mathcal{P}(O_\mathbf{s})}\max_{a\in A_\mathbf{s}}\sum_{o\in O_\mathbf{s}}\pi_o(R(\mathbf{s},a,o)+F(\mathbf{s}')) \tag{7.8a}$$

$$= \max_{\sigma\in\mathcal{P}(A_\mathbf{s})}\min_{o\in O_\mathbf{s}}\sum_{a\in A_\mathbf{s}}\sigma_a(R(\mathbf{s},a,o)+F(\mathbf{s}')). \tag{7.8b}$$

Therefore, let σ^* be the induced myopic strategy for the robber by (7.8b) and assume he plays according to σ^*. We study games where play stops after i steps. First, consider the game where the

robber obtains the payoff $GV(\mathbf{s})$ at the end of the ith step. Since GV is a fixpoint of T and σ^* is the myopic strategy with respect to GV the robber is assured at least a payoff of $GV(\mathbf{s}_0)$ by playing σ^*. Now consider the normal game truncated to i steps. If the game ends at step i it does so in a terminal state. Since $GV(\mathbf{s}) = 0$ for all terminal states $\mathbf{s}$ we know that the robber is assured at least $GV(\mathbf{s}_0)$ by playing σ^*. Therefore, let us assume the game does not end at step i. Let π be the strategy of the cops such that they can end the game with probability $p > 0$ after at most t steps. Since the cops could play π we know that the probability that the game would continue after i steps is at most $(1-p)^{\lfloor \frac{i}{t} \rfloor}$. Therefore, the payoff the robber achieves after i steps is at least

$$GV(\mathbf{s}_0) - (1-p)^{\lfloor \frac{i}{t} \rfloor} \max_{\mathbf{s} \in S} GV(\mathbf{s}).$$

When the game continues after the ith step the robber could, due to misplay, loose as much as the expected value of the game. Since the expected value of the game, due to the proof of Lemma 7.3, is

$$\mathbb{E}(GV(\mathbf{s})) \leq \frac{1}{p} t \|R\|$$

the robber can assure to obtain at least

$$GV(\mathbf{s}_0) - (1-p)^{\lfloor \frac{i}{t} \rfloor} \max_{\mathbf{s} \in S} GV(\mathbf{s}) - (1-p)^{\lfloor \frac{i}{t} \rfloor} \frac{1}{p} t \|R\|.$$

Since this holds true for any i, hence also for $i \to \infty$, we obtain that the robber can assure a payoff of at least $GV(\mathbf{s}_0)$.

It remains to prove that the cop can also assure that the payoff is at most $GV(\mathbf{s}_0)$. Let π^* be the induced myopic strategy due to (7.8a). To use the above method we have to show that the cop can win (as in Definition 7.2) with π^*. Assume he cannot. Then there is a strategy σ for the robber with which he can extend the play forever. However, after i steps the robber achieved at most

$$GV(\mathbf{s}_0) - \min_{\mathbf{s} \in S} GV(\mathbf{s}).$$

Since we know that the robber can extend game play forever we have that the above term goes to infinity as $i \to \infty$. This is not possible since $GV < \infty$ due to the assumption. Therefore, the cop can win with π^*. Analogously to the above we can prove that the cop can force the payoff to be at most

$$GV(\mathbf{s}_0) - (1-p)^{\lfloor \frac{i}{t} \rfloor} \min_{\mathbf{s} \in S} GV(\mathbf{s}) + (1-p)^{\lfloor \frac{i}{t} \rfloor} \frac{1}{p} t \|R\|$$

for some fixed t. Hence, for $i \to \infty$ we have that the cop can force the payoff to be at most $GV(\mathbf{s}_0)$. We proved that the myopic strategies σ^* and π^* with respect to GV achieve at least and at most a payoff of $GV(\mathbf{s}_0)$ when gameplay starts in $\mathbf{s}_0$. This holds for all $\mathbf{s}_0 \in S$. Hence, the two strategies define a Nash equilibrium and the equilibrium value of the game starting in $\mathbf{s}_0$ is $GV(\mathbf{s}_0)$. □

Lemma 7.8

Let GV^* be a fixpoint of T such that $0 \leq GV^* < \infty$. Let further GV_0 such that either of the following holds

(a) $GV_0 \leq GV^*$ and $GV_0 \leq T(GV_0)$.

(b) $GV_0 \geq GV^*$ and $GV_0 \geq T(GV_0)$.

Then, the value iteration with initial estimate GV_0 converges to a fixpoint of T.

Proof. Let us consider part (a). We prove by induction over i that for all $\mathbf{s}$, $GV_i(\mathbf{s}) \geq GV_{i-1}(\mathbf{s})$. Since $GV_0(\mathbf{s}) \leq T(GV_0)(\mathbf{s}) = GV_1(\mathbf{s})$ the induction base is due to the assumption. Therefore, let $GV_i(\mathbf{s}) \geq GV_{i-1}(\mathbf{s})$ for all $\mathbf{s}$. Due to Lemma 7.6 and the induction hypothesis for i we have

$$GV_{i+1}(\mathbf{s}) - GV_i(\mathbf{s}) = T(GV_i)(\mathbf{s}) - T(GV_{i-1})(\mathbf{s}) \geq 0,$$

which finishes the induction step. Hence, we know that the values in GV_i are always non-decreasing with i.

Furthermore, we can prove that $GV^*(\mathbf{s}) \geq GV_i(\mathbf{s})$. This holds for $i = 0$ due to the assumptions of this lemma. For every subsequent element we have, due to Lemma 7.6,

$$GV^*(\mathbf{s}) - GV_i(\mathbf{s}) = T(GV^*)(\mathbf{s}) - T(GV_{i-1})(\mathbf{s}) \geq 0.$$

That means that $GV_i(\mathbf{s})$ is bounded from above. Thus, we proved the point-wise convergence of $(GV_i)_i$. Since GV is defined over a finite set S we also have uniform convergence.

It remains to show that the value iteration also converges to a fixpoint of T. It is easy to see that T is continuous. Therefore, we have that $T(\lim_{i\to\infty} GV_i) = \lim_{i\to\infty} T(GV_i)$ and hence

$$T(GV')(\mathbf{s}) - GV'(\mathbf{s}) = \lim_{i\to\infty} GV_{i+1}(\mathbf{s}) - \lim_{i\to\infty} GV_i(\mathbf{s}) = 0.$$

This completes the proof of part (a). The proof of part (b) is analog by deriving that $GV^* \leq GV_{i+1} \leq GV_i$ which shows the convergence of $(GV_i)_i$. The above argument then completes the proof. □

Theorem 7.9

Let G be such that the cops can win the game (as in Definition 7.2). Let further GV^* be the optimal values of the game. If $GV_0(\mathbf{s}) = 0$ for all terminal states $\mathbf{s}$ and either of the following holds

(a) $GV_0 \leq GV^*$ and $GV_0 \leq T(GV_0)$

(b) $GV_0 \geq GV^*$ and $GV_0 \geq T(GV_0)$

then

$$\lim_{i\to\infty} GV_i = GV^* < \infty.$$

Proof. We have that $GV^* < \infty$ due to Lemma 7.3. Due to Lemma 7.5 we have that GV^* is a fixpoint of T. Convergence follows from Lemma 7.8. Correctness follows from Lemma 7.7. □

Corollary 7.10

Let G be a graph such that the cops can win the game. When setting GV_0 to any of the following, the value iteration converges.

(a) $GV_0 \equiv 0$.

(b) $GV_0(\mathbf{s}) = 0$ *for all terminal states* $\mathbf{s}$ *and* $GV_0(\mathbf{s}) = \infty$ *otherwise.*

Proof. Recall that we denote the reward function with R. Since $R \geq 0$ we can follow that $GV_1 \geq GV_0$. Part (a) now follows from Theorem 7.9.
Consider part (b). The value iteration does not change the value of terminal states. Hence, they remain at zero. Since $GV_0(\mathbf{s}) = \infty$ for all other states $\mathbf{s}$ the value can obviously only decrease the values in subsequent iterations, hence $G_1 \leq G_0$. Theorem 7.9 now completes the proof. □

The Markov Game formulation can also be used to approach the alternating game. Since a strategy in the alternating game corresponds to a pure strategy we only need to minimize over the action set instead of the probability distributions in (7.5a). Therefore, (7.5a) becomes

$$GV(\mathbf{s}) = \min_{o \in O_{\mathbf{s}}} \max_{a \in A_{\mathbf{s}}} Q(\mathbf{s}, a, o). \tag{7.9}$$

Since it does not involve linear programming but straightforward iteration over the action sets, (7.9) is much faster to solve than (7.5a). Note further, when playing with at least k cops on a k-cop-win graph the cops can win the game as in Definition 7.2. Hence, for the alternating version, the value iteration converges to the true values of the game and induces a Nash equilibrium.

When used for the alternating game, value iteration works similarly to the algorithm proposed by Hahn and McGillivray [33] (see Section 3.1). Both algorithms iterate over the entire state space, Hahn's algorithm over the state space with turn tag and value iteration over the state space without turn tag. In each iteration, value iteration assigns the current optimal value to the state. Hahn's algorithm checks if the current value of the state is different from infinity (the initialization is zero for all terminals and infinity for all other states). If not, it is set to the current optimal value, possibly yielding infinity again. Otherwise, the state is disregarded. Since the RA algorithm in Section 3.1 is an improvement of Hahn's algorithm, RA is to be preferred over value iteration when solutions are computed for the alternating move game.

When initialized with zero everywhere the value iteration subsequently increases the values of the state space (due to Lemma 7.6) until the final values are reached. Hence, one immediate idea to speed up convergence is to initialize the state space with an approximation on the true values. We have studied three methods. First, it is possible to use an heuristic function, i.e. an estimate on the value of the game. In the experiments (see Section 7.2) we used the forward heuristic that was used to

solve the alternating game more quickly in Chapter 4. Second, the values can be initialized with the optimal values from the alternating game. This is a plausible initialization since the alternating game can be solved much more quickly than the simultaneous game. The third method uses abstractions.

Definition 7.11 (abstraction)
Given a graph G, a graph G' is called an *abstraction* of G if there is a homomorphism $h : G \to G'$ and there are no more edges in G' than required by h.

Within our experiments we used octile maps together with clique abstraction [76, 77]. We compute the values of the game subsequently on every level of the abstraction hierarchy beginning with the highest. Given the values of the game on level l all the states on level $(l-1)$ that abstract into the same state in level l will be given the parent's value. Then, the game on level $(l-1)$ is solved with value iteration and the process continues until a solution is computed for the lowest level of abstraction, i.e. the base level. Experimental results for this kind of initialization can be found in Section 7.2.

There are other methods than value iteration to compute optimal policies that obey system (7.5). Littman [48] proposed to apply Q-learning known from regular Markov Decision Processes. The algorithm traverses the state space multiple times starting from a given start state and updates the values of Q while doing so. In contrast to value iteration, it therefore performs asynchronous updates on GV and Q and is only expected to have accurate values along the optimal playout originating in the given initial position. Within this chapter we are interested in solving the entire state space and therefore our focus is on value iteration only. The performance of the Q-learning approach and other possible methods for computing optimal policies for a given initial position would be interesting but we limit the study of the simultaneous game to this chapter. Hence, how to efficiently solve the simultaneous game for only one problem instance remains open.

7.2 Experiments

In this section, we are interested in the performance of value iteration when used to solve the simultaneous action game. Therefore, we will study the changes in the number of iterations when using the three different initialization methods from Section 7.1. Furthermore, we will introduce a measure of similarity of two strategies and measure how this similarity develops over the course of subsequent iterations.

Within our implementation of the value iteration we solve each Bellman equation (7.5) with the algorithm by Robinson [70]. This algorithm solves a matrix game by computing the probability vectors over the actions of both players. This is done iteratively, running playouts with the current estimates and updating the estimates according to these playouts. We run the iterative process until the payoff for the cop and the robber only differ by a parameter ϵ, i.e. until the difference of the left and right terms of (7.2) is within ϵ. The matrix game for each state in the value iteration is defined

by $Q(\mathbf{s})$ in (7.5b).

While running the value iteration we keep track of the difference of the current estimates GV_i. We stop iterating when

$$\|GV_{i+1} - GV_i\| \leq \delta$$

where $\|\cdot\|$ is the maximum norm and δ is a given parameter and is a mean to control the accuracy of the results.

The runtime of the algorithm highly depends on the structure of the map since this determines the structure of the LP's induced by the Bellman equations. Therefore, we only study one map in these experiments. Furthermore, we want to investigate if it is possible to gain performance for the value iteration when using different methods of initialization of the state space. Since gains in performance are expected to be the highest on graphs that induce some regular structure we test on a 5×5 grid with octile connections here. For the definition of an octile map refer to Definition 2.8.

It is easy to see that one cop can eventually win the game, i.e. that the expected value of the game is finite for any initial position of cop and robber. Hence, we only experiment with one cop here. Recall, that the state space has $n^2 = 5^2 * 5^2 = 625$ states.

We initialize the state space with the following methods. First, the maximum norm distance metric, that measures the maximal distance between the coordinates of the cop's and the robber's position, is used to assign a value to every state. The second method assumes the robber will stand still and the cop runs towards him. This method returns an estimated value of such a playout, beginning in the respective state, by estimating the distance traveled by the cop with the maximum norm distance metric. Note that this is the forward heuristic used in Section 4.5. Third, we initialize every state with the value of the alternating game starting in this state. Finally, we use the solution to the simultaneous game on an abstraction.

The fourth method uses a hierarchy of clique abstractions [76] and computes a fixed, given number of iterations on each level of abstraction. Computation starts at the highest level of abstraction. To initialize the value of a state on the next lower level, the value of the abstract state on the higher level is scaled by a factor of two (which is the average graph distance of the states abstracted into two adjacent abstract states). After reaching the lowest level of abstraction, i.e. the original graph, we continue to run value iterations until the above criterion is satisfied. The number of iterations on each level was set to 10 during the following experiments.

We measured the difference $\|GV_{i+1} - GV_i\|$ from one iteration to the next for all methods after the initialization has happened. We set $\delta = 0.01$ and $\epsilon = 0.001$. The results are depicted in Figure 7.1 with logarithmic scale on the ordinate. The graphs are very similar for all methods of initialization except for initialization via abstractions, making them non-distinguishable. Hence, the number of iterations required to push the difference, of one iteration to the next, below δ is nearly the same for all methods of initializations. Only initializing by exploiting the abstraction hierarchy gains a small advantage. However, this method computed 10 iterations on each of two levels of

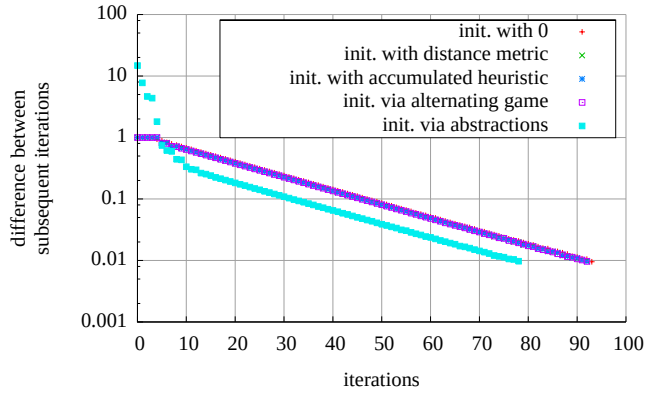


Figure 7.1: Convergence of $\|GV_{i+1} - GV_i\|$ of value iteration after the four different initializations.

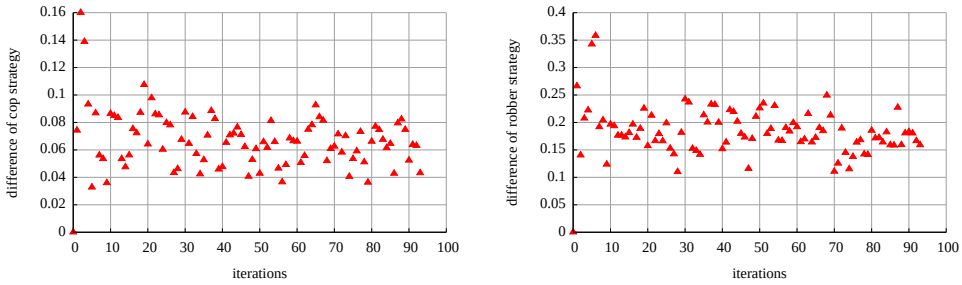
abstraction beforehand which is a computational overhead. To justify this overhead the number of iterations would have to decrease by a larger factor.

We encountered the same trends for different graphs as well. There are several reasons why these methods of initializations do not gain speedups. First, when the distance metric, accumulated heuristic or the alternating game are used for initialization, the assigned values are too small to make a big difference. For example, when the cop and robber are in positions (1,1) and (3,3) (the indices begin at 0) the distance metric is 2, the accumulated heuristic is 3 and the optimal value for the alternating game is 5. However, the expected value of the simultaneous action game starting in these vertices is greater than 21. Hence, there are still many iterations required to iteratively increase the values homogeneously over the state space. When the abstraction hierarchy is used, we have locally constant values in both dimensions of the state space after the initialization. Subsequent iterations are used to change these locally constant regions and to converge to a real solution.

When plotting the values of the state space for octile connected 1-cop-win graphs after each iteration we found that they have the same structure as the converged solution, only they are scaled by a factor. Hence, it seemed likely that for computing the strategies the value iteration did not have to converge to its true values. Therefore, we wanted to measure the difference of induced strategies of subsequent iterations. Recall that a strategy is a function that assigns a vector of probabilities to each state. Given two myopic policies for iteration i and $(i+1)$ we measure the difference between two corresponding strategies π_i and π_{i+1} in each state $\mathbf{s}$ by $\|\pi_{i+1}(\mathbf{s}) - \pi_i(\mathbf{s})\|$ where $\|\cdot\|$ is the maximum norm. That means we are measuring the maximum difference in the weights of these two strategies in each state. To have a global measure of similarity of the two strategies we compute the average of these differences, hence

$$\frac{1}{|S|}\sum_{\mathbf{s}\in S}\|\pi_{i+1}(\mathbf{s}) - \pi_i(\mathbf{s})\|.$$

We plot this measure for the value iteration with initialization of zero everywhere in Figure 7.2. The values for different initializations are similar and are therefore omitted here.



(a) cop (b) robber

Figure 7.2: The difference of the strategy for the cop and robber over the course of iterations.

These values show that the differences in the strategies are drastic, especially for the robber. However, this can be caused by the way the LPs in (7.5) are solved by Robinson's algorithm [70]. There might be multiple possible solutions to the induced linear programs for each state in the Bellman equations. However, our algorithm can only select one. The fact that the difference for the robber is a magnitude higher than for the cop suggests that the algorithm prefers certain minima/maxima of the linear programs that induce high differences in the strategies induced by subsequent iterations. The question, whether these differences can be consistently decreased over the course of iterations by selecting different solutions to the linear programs with a different LP solver remains open. Furthermore, another interesting question is how successful the induced strategies after each iteration are. This would have to be measured by computing a best response, which can be done by fixing one of the strategies in the Bellman equations (7.5). However, a similar exploitability does not necessarily mean similarity in strategy, i.e. how the agents move. Again, this study remains for future investigation.

Since the computation of a solution highly depends on the LP solver used we can only give approximate computation time results. It is likely that computation can be sped up when using a better LP solver. We computed the solution to the 7x7 octile connected grid with one cop with $\delta = 0.01$ and $\epsilon = 0.001$. Recall, that the state space has $7^2 * 7^2 = 2401$ states. The computation with initialization of zero everywhere converged after 218 iterations and took 19m 51s[1].

7.3 Summary and open problems

We introduced the simultaneous action game and its formulation as a linear program (LP). Unfortunately, due to its exponential growth, this LP is too large to be practically solvable. We proposed a different formulation of the simultaneous action game due to an undiscounted Markov Game formulation. The resulting Bellman equations can be solved with value iteration. We proved that, under the assumption that the expected value of the game is finite and the cops can hence win the game, the value iteration converges to the optimal values. Additionally we studied different methods of

[1] All experiments were run on a Intel Xeon© 2.5GHz CPU with 16GB RAM.

initializing the state space which unfortunately proved to gain no significant speedups. Furthermore, since value iteration has to iterate over the entire state space and solve the Bellman equations in each state, the algorithm is only feasible for relatively small graphs, in contrast to the RA algorithm for the alternating game. It remains an open problem whether the strategies induced by subsequent iterations are similar or not, i.e. if the value iteration can be terminated early if we only need an approximation to the optimal strategies rather than the optimal values of the game. Furthermore, it seems likely that our method can be improved in speed by using different LP solvers.

Conclusions

Pursuit games have been studied in many domains such as warfare, law enforcement and computer games. The game of cops and robber or moving target search is one special kind of pursuit game that we studied in this thesis.

In the first part of this work we investigated algorithms that compute optimal solutions to the game. Here, optimality means a Nash equilibrium. We outlined the retrograde analysis (RA) algorithm used to compute solutions for the entire state space, i.e. for every tuple of possible initial positions. This algorithm can also be used to compute a best response against a deterministic robber algorithm that does not keep track of the history of the game. Furthermore, we outlined possible optimizations when computing solutions where multiple cops are involved. We studied the theoretical runtime of the RA algorithm and evaluated its practical performance.

Additionally, we studied algorithms that compute solutions for given initial positions. Here, heuristics can be used to speed up computation. We improved the algorithm Two-Agent IDA* (TIDA*) by incorporating intelligent caching methods and enhanced Proof-Number Search with iterative bound increase similar to TIDA*. Moreover, we developed a new algorithm, called Reverse Minimax A* (RMA*), that computes solutions in a bottom-up fashion. Experiments conducted on multiple sets of maps and with one or two cops showed that our new method outperforms the previously known algorithms.

Although they scale well with respect to the map size, optimal algorithms still exceed modern computer game time constraints. Therefore, we investigated sub-optimal algorithms in the second part of this thesis. Hence, our goal was to compute very good solutions quickly. We discussed the existing approaches Cover, Dynamic Abstract Minimax (DAM), Minimax, a random beacon algorithm, hill climbing due to distance heuristic and developed two new methods, TrailMax and Dynamic Abstract TrailMax (DATrailMax). Conducting experiments where the algorithms competed against a Nash equilibrium playing cop were used to evaluate the algorithms' suboptimality. Additionally, we computed best responses against some of the algorithms to assess their exploitability. These experiments showed that our new methods outperform the existing approaches in quality while meeting modern computer game time constraints. Hence, this work redefines the state-of-the-art in perfect information moving target search. Furthermore, detailed analysis showed that previous results on the Cover heuristic were flawed but that our alternative variation can be used to circumvent

some of the problems.

We have outlined known mathematical results for the cop number, the search time of a graph, the characterization of cop-win graphs, the complexity for computing solutions to the cops and robber problem and other variations of the game. The capture time is a new parameter that has not been studied in the literature before and is important for the algorithmic study of the game. We established bounds on the capture time on cop-win graphs and gained some results on the characterization of some 2-cop-win graphs. We further engaged in a discussion on possible characterization of general k-cop-win graphs.

As another by-product of the main work on this thesis, we investigated methods to solve the simultaneous action game. Here, Nash equilibria are to be found in mixed strategies and therefore the previous game tree search algorithms are not a feasible approach. We expressed the game as an undiscounted Markov Game formulation and used value iteration to solve it. Furthermore, we proved convergence and correctness of the algorithm for certain initial conditions. Unfortunately, our ideas to speed up convergence with different initializations did not gain significant improvements.

Bibliography

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

[2] L. V. Allis, M. van der Meulen, and H. J. van den Herik. Proof-number search. *Artificial Intelligence*, 66:91–124, 1994.

[3] L. Alonso and E. M. Reingold. Lower bounds for cops and robber pursuit. *submitted*, 2008.

[4] B. Alspach. Searching and sweeping graphs: A brief survey. *Le Matematiche*, 59:5–37, 2006.

[5] T. Andreae. Note on a pursuit game played on graphs. *Discrete Applied Mathematics*, 9:111–115, 1984.

[6] T. Andreae. On a pursuit game played on graphs for which a minor is excluded. *Journal of Combinatorial Theory*, Ser. B 41:37–47, 1986.

[7] R. P. Anstee and M. Farber. On bridged graphs and cop-win graphs. *Journal of Combinatorial Theory, Series B*, 44(1):22–28, 1988.

[8] A. Berarducci and B. Intrigila. On the cop number of a graph. *Advances in Applied Mathematics*, 14:389–403, 1993.

[9] F. Berger, A. Grüne, and R. Klein. How many lions can one man avoid? Technical report, Rheinische Friedrich-Wilhelms-Universität Bonn, November 2007.

[10] M. Bern, E. L. Lawler, and A. L. Wong. Linear-time computation of optimal subgraphs of decomposable graphs. *Journal of Algorithms*, 8(2):216–235, June 1987.

[11] A. Bonato and E. Chiniforooshan. Pursuit and evasion from a distance: algorithms and bounds. *Proceedings of the Fifth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 1–10, January 3 2009.

[12] A. Bonato, G. Hahn, P. Golovach, and J. Kratochvil. The search-time of a graph. *Discrete Mathematics*, 2008.

[13] A. Bonato, G. Hahn, and C. Wang. The cop density of a graph. *Contributions to Discrete Mathematics*, 2(2):133–144, 2007.

[14] A. Bonato, P. Prałat, and C. Wang. Pursuit-evasion in models of complex networks. *Internet Mathematics*, 4:419–436, 2007.

[15] V. Bulitko and N. Sturtevant. State abstraction for real-time moving target pursuit: A pilot study. *Proceedings of the National Conference on Artificial Intelligence (AAAI), Workshop on Learning For Search*, (72-79), 2006. Boston, Massachusetts.

[16] V. Chepoi. Bridged graphs are cop-win graphs: An algorithmic proof. *Journal of Combinatorial Theory*, B 69:97–100, 1997.

[17] E. Chiniforooshan. A better bound for the cop number of general graphs. *Journal of Graph Theory*, 58:45–48, 2008.

[18] N. E. Clarke. *Constrained Cops and Robber*. PhD thesis, Dalhousie University, 2002.

[19] N. E. Clarke. A witness version of the cops and robber game. *Discrete Mathematics*, 309(10):3292–3298, 2009.

[20] R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer-Verlag, Heidelberg, July 2005.

[21] F. V. Fomin, P. A. Golovach, A. Hall, M. Mihalák, E. Vicari, and P. Widmayer. How to guard a graph? *Algorithms and Computation*, 5369:318–329, 2008.

[22] F. V. Fomin, P. A. Golovach, and J. Kratochvíl. On tractability of cops and robbers game. *Fifth IFIP International Conference On Theoretical Computer Science*, 273:171–185, 2008.

[23] F. V. Fomin and D. Thilikos. An annotated bibliography on guaranteed graph searching. *Theoretical Computer Science*, 2008.

[24] P. Frankl. Cops and robbers in graphs with large girth and cayley graphs. *Discrete Applied Mathematics*, 17(3):301–305, 1987.

[25] P. Frankl. On a pursuit game on cayley graphs. *Combinatorica*, 7(1):67–70, 1987.

[26] R. Gasser. Applying retrograde analysis to nine men's morris. *Heuristic Programming in Artificial Intelligence*, pages 161–173, 1991.

[27] T. Gavenčiak. Games on graphs. Master's thesis, Department of Applied Mathematics, Charles University, Prague, 2007.

[28] T. Gavenčiak. Cop-win graphs with maximal capture-time. *Studentská vědecká a odborná činnost (SVOC)*, April 30 2008.

[29] M. Goldenberg, A. Kovarksy, X. Wu, and J. Schaeffer. Multiple agents moving target search. *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1536–1538, 2003.

[30] A. S. Goldstein and E. M. Reingold. The complexity of pursuit on a graph. *Theoretical Computer Science*, 143(1):93–112, 1995.

[31] G. Hahn. Cops, robbers and graphs. *Tatra Mountains Mathematical Publications*, 36(2):163–176, 2007.

[32] G. Hahn, F. Laviolette, N. Sauer, and R. Woodrow. On cop-win graphs. *Discrete Mathematics*, 258(1):27–41(15), 2002.

[33] G. Hahn and G. MacGillivray. A note on k-cop, l-robber games on graphs. *Discrete Mathematics*, 306(19-20):2492–2497, 2006.

[34] Y. Hamidoune. On a pursuit game on cayley digraphs. *European J. Combin.*, 8:289–295, 1987.

[35] R. Isaacs. *Differential Games: A Mathematical Theory With Applications to Warfare and Pursuit, Control and Optimization*. Dover Publications, New York, 1965. ISBN 978-0-486-40682-4.

[36] A. Isaza. A heuristic-based approach to multi-agent moving target search. Master's thesis, Department of Computing Science at the University of Alberta, 2008.

[37] A. Isaza, J. Lu, V. Bulitko, and R. Greiner. A cover-based approach to multi-agent moving target pursuit. *Artificial Intelligence and Interactive Entertainment Conference (AIIDE)*, 2008.

[38] T. Ishida and R. E. Korf. Moving target search. *International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 204–210, 1991.

[39] T. Ishida and R. E. Korf. Moving-target search: A real-time search for changing goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(6):609–619, 1995.

[40] V. Isler and N. Karnad. The role of information in the cop-robber game. *Theoretical Computer Science*, 399:179–190, 2008.

[41] G. Joret, M. Kamiński, and D. O. Theis. The cops and robber game on graphs with forbidden (induced) subgraphs. `http://arxiv.org/abs/0804.4145`, 2008.

[42] A. Kishimoto and M. Müller. A general solution to the graph history interaction problem. *Nineteenth National Conference on Artificial Intelligence (AAAI 2004)*, pages 644–649, 2004.

[43] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[44] S. Koenig, M. Likhachev, and X. Sun. Speeding up moving-target search. *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1136–1143, 2007.

[45] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[46] R. E. Korf. A simple solution to pursuit games. In *The 11th International Workshop on Distributed Artificial Intelligence*, pages 183–194, Glen Arbor, Mich., Feb 1992.

[47] H. J. Kushner and S. G. Chamberlain. Finite state stochastic games: Existence theorems and computational procedures. *IEEE Transactions on Automatic Control*, AC-14(3):248–255, 1969.

[48] M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. *Proceedings of the Eleventh International Conference on Machine Learning*, pages 157–163, 1994. San Francisco, CA.

[49] M. L. Littman and C. Szepesvári. A generalized reinforcement-learning model: Convergence and applications. *Proceedings of International Conference of Machine Learning '96*, pages 310–318, 1996.

[50] T. Łuczak and P. Prałat. Chasing robbers on random graphs: zigzag theorem. *Random Structures and Algorithms*, 2009.

[51] M. Maamoun and H. Meyniel. On a game of policemen and robber. *Discrete Appl. Math.*, 17(3):307–309, 1987.

[52] A. Martelli and U. Montanari. Optimizing decision trees through heuristically guided search. *Communications of the ACM*, 21(12):1025–1039, 1978.

[53] B. McKay. Collection of combinatorial data. `http://cs.anu.edu.au/~bdm/data/`.

[54] N. Megiddo, S. Hakimi, M. Garey, D. Johnson, and C. Papadimitriou. The complexity of searching a graph. *Journal of the Association for Computing Machinery*, 35(1):18–44, January 1988.

[55] S. Melax. New approaches to moving target search. *National Conference on Artificial Intelligence (AAAI), Fall Symposium*, pages 30–38, 1993.

[56] S. Melax. New approaches to moving target search. Master's thesis, Department of Computing Science at the University of Alberta, 1993.

[57] C. Moldenhauer and N. Sturtevant. Evaluating strategies for running from the cops. *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 584–589, 2009.

[58] C. Moldenhauer and N. Sturtevant. Optimal solutions for moving target search. *Autonomous Agents and Multiagent Systems (AAMAS)*, 2009.

[59] O. Morgenstern and J. von Neumann. *The Theory of Games and Economic Behavior*. Princeton University Press, 1947.

[60] A. Nagai. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. PhD thesis, Department of Information Science, University of Tokyo, 2002.

[61] P. J. Nahin. *Chases and Escapes: The Mathematics of Pursuit and Evasion*. Princeton University Press, Princeton, NJ, 2007.

[62] J. F. Nash. Non-cooperative games. *The Annals of Mathematics*, 54(2):286–295, 1951.

[63] N. Nisse and K. Suchan. Fast robber in planar graphs. *Proceedings of the 34th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2008)*, pages 312–323, 2008.

[64] R. Nowakowski and P. Winkler. Vertex-to-vertex pursuit in a graph. *Discrete Mathematics*, 43:235–239, 1983.

[65] T. Parsons. Pursuit-evasion in a graph. *Theory and Applications of Graphs*, pages 426–441, 1978.

[66] S. D. Patek and D. P. Bertsekas. Stochastic shortest path games. *SIAM Journal on Control and Optimization*, 37(3):804–824, 1999.

[67] A. E. Prieditis and E. Fletcher. Two-agent IDA*. *Journal of Experimental and Theoretical Artificial Intelligence*, 10(4):451–485, 1998.

[68] A. Quillot. *Etudes de quelques problèmes sur les graphes et hypergraphes et applications à la théorie des jeux à information complète*. PhD thesis, Université de Paris VI, 1978.

[69] A. Quillot. Discrete pursuit games. *Procedings of the 13th Conference on Graphs and Combinatorics*, 1982.

[70] J. Robinson. An iterative method of solving a game. *Annals of Mathematics*, 44(2):296–301, 1951.

[71] J. Schaeffer, N. Burch, Y. Bjornsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, pages 1518–1522, 2007.

[72] B. Schroeder. The copnumber of a graph is bounded by $\lfloor \frac{3}{2}\text{genus}(g) \rfloor + 3$. *Trends Math.*, pages 243–263, 2001.

[73] P. Seymour and R. Thomas. Graph searching and a min-max theorem for tree-width. *Journal of Combinatorial Theory Series B*, 58(1):22–33, May 1993.

[74] L. S. Shapley. Stochastic games. *Proceedings of the National Academy of Sciences, Mathematics*, 39(10):1095–1100, October 1 1953.

[75] Y. C. Stamatiou and D. M. Thilikos. Monotonicity and inert fugitive search games. Technical report, Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Barcelona, Spain, 2003.

[76] N. Sturtevant and M. Buro. Partial pathfinding using map abstraction and refinement. *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 1392–1397, 2005.

[77] N. Sturtevant and R. Jansen. An analysis of map-based abstraction and refinement. *Seventh Symposium on Abstraction, Reformulation and Approximation*, pages 344–358, 2007. Whistler, BC.

[78] C. Szepesvári and M. L. Littman. Generalized markov decision processes: Dynamic-programming and reinforcement-learning algorithms. Technical Report CS-96-11, Brown University, Providence, 1996.

[79] B. Thériault. Reconnaissance des graphes policier-gagnants. Master's thesis, Université de Montréal, 2006.

[80] K. Thompson. Retrograde analysis of certain endgames. *Journal of the International Computer Chess Association*, 9(3):131–139, 1986.

Appendix A

Properties of octile maps

Lemma A.1

Let G be a cop-win octile map, $d(v,w)$ be the graph distance between vertices $v, w \in G$. Let $B(v,d) = \{w \in V(G) | d(v,w) \leq d\}$ be the ball around vertex v with radius d. Let $\partial B(v,d) = \{w \in V(G) | d(v,w) = d\}$ be the boundary of $B(v,d)$. For $S \subsetneq G$ denote the set of vertices in S that have minimal distance to v by $\perp_S (v) = \{w \in S | d(v,w) = \min_{w' \in S} d(v,w')\}$.
Then, for any $v \in G$ and $w \notin B(v,d)$ we have that $\perp_{\partial B(v,d)} (w)$ is connected and all its vertices are on a horizontal or vertical line segment of ∂B.

Proof. Consider Figure A.1 for a visualization of the definitions in the lemma. Fix a radius d and the vertices $v \in G, w \in G - B(v,d)$. For ease of reading we assign $B = B(v,d)$, $\partial B = \partial B(v,d)$, $\perp = \perp_{\partial B(v,d)} (w)$. Recall that the length of a path is the number of its vertices reduced by one.
First, notice that a shortest path from w to a vertex $x \in \perp$ does not have any common vertices with B except for x. Otherwise, the path goes through B and enters B before reaching x. Let $y \in \partial B$ be the entrance vertex of the path into B. Then $d(w,y) < d(w,x)$ which is a contradiction to the minimality of the distance to w in $\perp$.
We will show that $\perp$ is a subset of one connected component of ∂B. Assume this is not the case. Then, there exist vertices $x, y \in \perp$ that are in different components of ∂B. Recall that the vertices of an octile map are located on the $\mathbb{N} \times \mathbb{N}$ grid. Therfore, the boundary ∂B of B is only disconnected in places where a grid point is not included into the graph as a vertex. Find a shortest path p_{wx} and p_{wy} from w to x and w to y, respectively. These paths do not go through B. Find a path p_{xy} from x

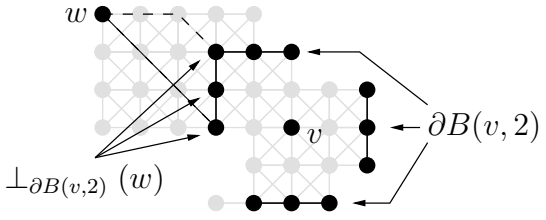


Figure A.1: Depiction of the definitions in Lemma A.1

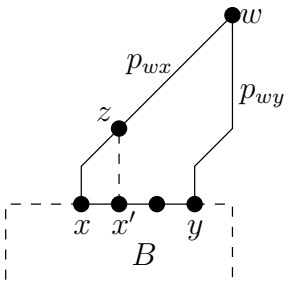


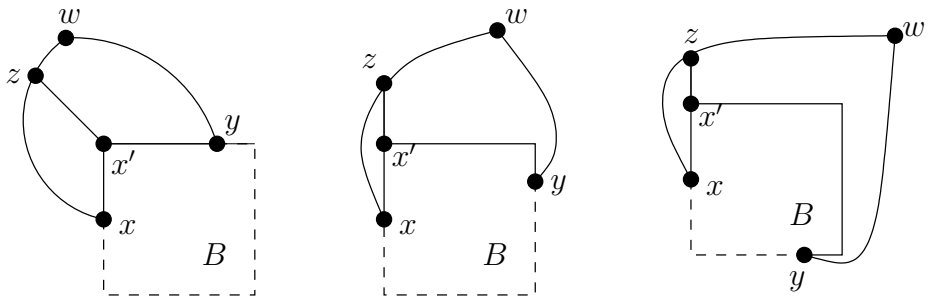
Figure A.2: Constructing shortest paths for vertices in between x and y.

to y through vertices of B. Construct a cycle through w, x and y with the paths p_{wx}, p_{xy}, p_{wy}. This cycle surrounds a grid point that disconnects ∂B. Hence, there is a, possibly shorter, induced cycle in the graph around this grid point that has no bridges. Since the smallest such cycle in an octile map is of size eight, the graph is not 1-cop-win. This is in contradiction to the assumption of the lemma and therefore $\perp$ is in one connected component of ∂B.

We will now introduce what will be called the intersection technique in the following. Consider Figure A.2 for an example. Given two vertices $x, y \in \perp$ we will specify a vertex x' in the same component of ∂B and a line direction. In the example in Figure A.2 we chose a vertical line. Furthermore, we consider shortest paths p_{wx} and p_{wy} from w to x and w to y, respectively. The given line intersects in a grid point with at least one of the two paths at least once. Let z be the intersection grid point that is closest to x'. Since z is on one of the paths it is also a vertex. Furthermore, the grid points from x' to z along the specified line direction are vertices of the graph and the edges in between them are edges of the graph. Otherwise, the paths p_{wx}, the path from x to y through vertices of ∂B and p_{wy} form a cycle that encloses a grid point that is not a vertex of the graph. By the above argument, this is not possible in 1-cop-win octile maps. Let $p_{zx'}$ be the path formed by the vertices between z and x' along the specified line direction. Furthermore, without loss of generality, let z be on p_{wx}. Let p_{wz} denote the part of p_{wx} from w to z and p_{zx} denote the part of p_{wx} from z to x.

We now show that all vertices of $\perp$ are on a horizontal or vertical line segment along ∂B. This is proved by contradiction. Hence, assume that there is $x, y \in \perp$ that are not on the same line segment of ∂B. We have plotted the three cases that can occur, up to symmetry, in Figure A.3. Note that alternative positions of w and in particular the paths p_{wx} and p_{wy} can be rejected with the above argument of constructing a cycle around a non included grid point.

Consider the first case depicted in Figure A.3(a). We will set x' to be the corner vertex of the two different line segments of ∂B that x and y are on. The line direction is diagonal. Then, $p_{zx'}$ is shorter than p_{zx}. This is because p_{zx} has to cross the line through x' and y to reach x. For example, in Figure A.3(a) the vertical coordinate coming from z has to drop below the vertical coordinate of x'. This requires at least as many edges as there are in $p_{zx'}$. Thus, we can construct a new path from w to x', via the paths p_{wz} and $p_{zx'}$, that is shorter than p_{wx}. Since $x' \in \partial B$ this yields a



(a) x and y one corner appart

(b) x and y two corners appart

(c) x and y three corners appart

Figure A.3: Three different cases when $x, y \in \perp$ are in different line segments of ∂B and their constructions of contradiction.

contradiction to the minimality of $x \in \perp$.

Consider the second case depicted in Figure A.3(b). We set x' as depicted and choose the line direction to be horizontal. Analog to the first case we obtain that $p_{zx'}$ is shorter than p_{zx}. Hence, we can construct a new path from w to x' that is shorter than p_{wx} which yields a contradiction. The third case depicted in Figure A.3(c) follows analog.

We have shown that all vertices of $\perp$ are on the same component of ∂B and are all on a vertical or horizontal line segment of ∂B. It remains to show that $\perp$ is connected. Therefore, let $x, y \in \perp$ be two vertices that are not adjacent. Consider Figure A.2 for an illustration. Again, we use the intersection technique by choosing x' to be the neighbor of x towards y and a perpendicular direction to the xy direction. Then $p_{zx'}$ is as long as p_{zx}. Hence, the path obtained by combining p_{wz} and $p_{zx'}$ is as long as p_{wx} and therefore $x' \in \perp$. By induction we obtain that all vertices in between x and y are in $\perp$. Hence, $\perp$ is connected.

This completes the proof.

□

Appendix B

Suboptimality experiments

algorithm	optim.				nodes expanded per call				nodes touched per call				nodes expanded per turn				nodes touched per turn			
	< 1k	≥ 1k	≥ 2k	≥ 5k	< 1k	≥ 1k	≥ 2k	≥ 5k	< 1k	≥ 1k	≥ 2k	≥ 5k	< 1k	≥ 1k	≥ 2k	≥ 5k	< 1k	≥ 1k	≥ 2k	≥ 5k
original Cover	69.62	66.41	65.12	57.74	8.517	8.496	8.583	8.378	86.111	87.637	91.033	89.722								
improved Cover	92.75	91.27	90.71	87.31	3.778	3.825	4.054	4.100	96.199	99.089	108.470	109.245								
improved Greedy	78.69	78.16	77.56	74.24	0.002	0.001	0.000	0.000	0.013	0.006	0.003	0.001								
Greedy/Minimax(1)	79.74	79.12	79.02	75.69	0.002	0.001	0.000	0.000	0.013	0.006	0.003	0.001								
Minimax(3)	83.79	82.10	81.30	77.42	0.043	0.019	0.010	0.003	0.296	0.131	0.064	0.018								
Minimax(5)	85.95	83.74	82.34	78.05	0.315	0.144	0.072	0.020	2.288	1.019	0.505	0.137								
Minimax(7)	87.89	84.90	83.24	78.63	1.368	0.655	0.337	0.095	10.207	4.794	2.445	0.661								
Minimax(9)	89.47	86.14	84.20	79.09	4.277	2.159	1.149	0.331	32.330	16.064	8.491	2.336								
Minimax(11)	90.94	87.30	85.03	79.58	10.655	5.633	3.104	0.912	81.238	42.425	23.276	6.502								
DAM(1,1)	84.68	83.28	83.43	81.52	0.030	0.025	0.022	0.020	0.206	0.173	0.155	0.138								
DAM(1,3)	87.58	87.60	87.93	87.86	0.112	0.061	0.039	0.023	0.686	0.376	0.247	0.153								
DAM(1,5)	85.66	86.69	86.99	86.22	0.536	0.263	0.151	0.053	3.271	1.562	0.892	0.313								
DAM(1,7)	85.41	85.83	86.23	84.77	1.692	0.822	0.459	0.139	10.776	5.051	2.783	0.808								
DAM(1,9)	84.02	84.28	85.05	83.94	4.385	2.106	1.189	0.334	28.474	13.274	7.338	1.948								
DAM(1,11)	84.00	83.80	84.41	83.19	9.256	4.529	2.572	0.713	60.426	28.777	15.961	4.137								
DAM(0.75,1)	88.57	87.90	88.17	85.64	0.013	0.010	0.008	0.006	0.092	0.071	0.056	0.039								
DAM(0.75,3)	88.03	88.62	89.24	88.86	0.072	0.040	0.026	0.014	0.463	0.259	0.169	0.095								
DAM(0.75,5)	86.04	87.09	87.76	87.57	0.385	0.187	0.105	0.038	2.473	1.162	0.650	0.232								
DAM(0.75,7)	85.55	86.17	86.81	86.09	1.296	0.621	0.337	0.102	8.624	3.956	2.110	0.610								
DAM(0.75,9)	84.25	84.53	85.48	85.04	3.479	1.642	0.911	0.253	23.528	10.725	5.800	1.515								
DAM(0.75,11)	84.07	83.99	84.69	83.96	7.510	3.620	2.020	0.549	51.024	23.816	12.908	3.267								
DAM(0.5,1)	88.21	87.28	87.29	84.05	0.008	0.005	0.003	0.002	0.052	0.036	0.023	0.013								
DAM(0.5,3)	90.12	90.04	90.17	87.55	0.062	0.031	0.018	0.007	0.407	0.203	0.116	0.047								
DAM(0.5,5)	88.63	89.23	89.71	88.41	0.347	0.160	0.085	0.027	2.292	1.023	0.540	0.165								
DAM(0.5,7)	87.48	87.93	88.79	87.95	1.220	0.555	0.293	0.084	8.281	3.618	1.876	0.507								
DAM(0.5,9)	86.46	86.61	87.75	87.32	3.320	1.536	0.823	0.224	22.800	10.168	5.314	1.343								
DAM(0.5,11)	85.82	85.60	86.85	86.55	7.388	3.505	1.884	0.505	50.696	23.266	12.147	3.001								
DAM(0.25,1)	83.27	82.90	83.09	80.17	0.003	0.002	0.002	0.001	0.023	0.015	0.012	0.006								
DAM(0.25,3)	87.92	87.26	87.52	84.01	0.051	0.025	0.014	0.005	0.340	0.162	0.093	0.034								
DAM(0.25,5)	88.88	88.22	88.63	85.42	0.307	0.137	0.070	0.020	2.157	0.929	0.467	0.127								
DAM(0.25,7)	89.45	88.77	88.97	85.75	1.195	0.530	0.266	0.070	8.691	3.725	1.832	0.448								
DAM(0.25,9)	89.48	88.97	89.17	86.33	3.531	1.605	0.812	0.202	26.179	11.582	5.723	1.319								
DAM(0.25,11)	89.53	88.89	89.08	86.37	8.333	3.935	2.008	0.484	62.389	28.784	14.392	3.183								
DAM(0,1)	equivalent to Greedy/Minimax(1)																			
DAM(0,3)	85.41	84.60	84.52	80.32	0.047	0.022	0.012	0.004	0.319	0.144	0.076	0.026								
DAM(0,5)	86.88	85.77	85.82	81.55	0.314	0.142	0.072	0.020	2.259	1.000	0.500	0.136								
DAM(0,7)	88.51	87.21	87.31	83.11	1.330	0.622	0.320	0.088	9.838	4.524	2.306	0.611								
DAM(0,9)	89.18	87.89	87.90	83.67	4.150	2.039	1.088	0.301	31.169	15.127	8.031	2.132								
DAM(0,11)	90.03	88.69	88.62	84.59	10.306	5.292	2.910	0.822	78.118	39.737	21.825	5.902								
DAM1(1,1)	84.12	82.63	83.42	81.13	0.032	0.023	0.020	0.015	0.211	0.157	0.137	0.102								
DAM1(1,3)	90.45	89.12	88.62	87.19	0.101	0.054	0.032	0.016	0.609	0.322	0.198	0.103								
DAM1(1,5)	90.78	89.63	89.13	86.38	0.516	0.251	0.140	0.044	3.155	1.484	0.823	0.253								
DAM1(1,7)	89.14	87.61	87.03	84.18	1.678	0.816	0.451	0.131	10.739	5.022	2.732	0.752								
DAM1(1,9)	88.71	86.85	86.52	83.45	4.355	2.089	1.181	0.328	28.408	13.194	7.298	1.899								
DAM1(1,11)	88.61	86.31	85.77	82.75	9.162	4.505	2.550	0.701	60.054	28.684	15.829	4.058								
DAM1(0.75,1)	88.20	87.48	87.92	85.18	0.014	0.010	0.008	0.006	0.094	0.072	0.057	0.039								
DAM1(0.75,3)	90.60	89.88	89.54	87.84	0.062	0.033	0.019	0.009	0.395	0.207	0.125	0.059								

DAM1(0.75,5)	90.88	89.88	89.53	87.39	0.368	0.176	0.096	0.030	2.374	1.088	0.586	0.175							
DAM1(0.75,7)	89.22	87.76	87.47	85.01	1.278	0.612	0.328	0.094	8.556	3.909	2.053	0.550							
DAM1(0.75,9)	88.81	86.90	86.54	83.91	3.452	1.623	0.901	0.244	23.466	10.632	5.741	1.453							
DAM1(0.75,11)	88.68	86.46	85.92	83.12	7.428	3.597	1.992	0.536	50.696	23.731	12.736	3.175							
DAM1(0.5,1)	88.18	87.23	87.24	84.03	0.007	0.005	0.003	0.002	0.049	0.033	0.022	0.013							
DAM1(0.5,3)	92.43	91.32	90.66	87.31	0.055	0.026	0.014	0.005	0.358	0.169	0.093	0.033							
DAM1(0.5,5)	92.25	91.25	90.47	87.81	0.335	0.154	0.081	0.024	2.223	0.979	0.511	0.143							
DAM1(0.5,7)	90.12	89.14	88.47	86.28	1.208	0.553	0.291	0.082	8.247	3.612	1.856	0.486							
DAM1(0.5,9)	89.58	87.89	87.44	85.05	3.303	1.542	0.828	0.226	22.802	10.260	5.347	1.346							
DAM1(0.5,11)	89.12	87.06	86.56	84.01	7.298	3.495	1.890	0.509	50.287	23.282	12.196	3.014							
DAM1(0.25,1)	83.27	82.90	83.08	80.17	0.003	0.001	0.001	0.000	0.020	0.011	0.006	0.003							
DAM1(0.25,3)	88.10	87.01	86.95	83.81	0.046	0.021	0.011	0.003	0.310	0.139	0.071	0.022							
DAM1(0.25,5)	89.97	88.47	88.23	84.87	0.305	0.137	0.069	0.020	2.161	0.935	0.463	0.125							
DAM1(0.25,7)	90.97	89.07	88.27	84.80	1.222	0.550	0.275	0.073	8.954	3.886	1.895	0.471							
DAM1(0.25,9)	91.67	89.33	88.41	84.71	3.614	1.681	0.852	0.217	26.975	12.180	6.022	1.411							
DAM1(0.25,11)	92.12	89.51	88.40	84.47	8.546	4.100	2.102	0.520	64.444	30.155	15.120	3.408							
DAM1(0,1-11)	equivalent to Minimax(1-11)																		
RandomBeacons(40,1)	69.45	66.97	65.58	65.17	0.173	0.167	0.156	0.130	1.277	1.175	1.084	0.880							
RandomBeacons(40,2)	70.99	68.11	66.39	65.49	0.174	0.169	0.156	0.130	1.287	1.183	1.084	0.880	0.100	0.092	0.083	0.067	0.742	0.647	0.575	0.452
RandomBeacons(40,3)	72.49	69.10	67.04	65.92	0.175	0.170	0.156	0.130	1.290	1.190	1.084	0.881	0.077	0.067	0.059	0.046	0.568	0.474	0.407	0.310
RandomBeacons(40,4)	73.70	70.08	67.76	66.25	0.176	0.170	0.156	0.130	1.294	1.195	1.085	0.882	0.066	0.055	0.046	0.035	0.484	0.388	0.324	0.239
RandomBeacons(40,5)	74.91	70.94	68.36	66.61	0.176	0.171	0.156	0.131	1.296	1.200	1.085	0.882	0.059	0.048	0.039	0.029	0.435	0.338	0.274	0.197
RandomBeacons(40,6)	75.91	71.68	68.94	66.89	0.177	0.172	0.156	0.130	1.299	1.208	1.087	0.882	0.055	0.043	0.035	0.025	0.404	0.306	0.241	0.169
RandomBeacons(40,7)	76.73	72.38	69.50	67.18	0.177	0.173	0.157	0.131	1.301	1.214	1.089	0.883	0.052	0.040	0.031	0.022	0.384	0.283	0.219	0.148
RandomBeacons(40,8)	77.41	72.95	70.01	67.48	0.178	0.174	0.157	0.131	1.303	1.220	1.091	0.884	0.050	0.038	0.029	0.020	0.370	0.268	0.202	0.133
RandomBeacons(40,9)	78.14	73.57	70.48	67.87	0.178	0.175	0.157	0.131	1.304	1.223	1.094	0.884	0.049	0.036	0.027	0.018	0.359	0.255	0.190	0.122
RandomBeacons(40,10)	78.75	74.06	70.98	68.14	0.178	0.175	0.158	0.131	1.304	1.228	1.096	0.884	0.048	0.035	0.026	0.017	0.351	0.245	0.179	0.112
RandomBeacons(40,11)	79.31	74.49	71.38	68.42	0.178	0.176	0.158	0.131	1.301	1.234	1.100	0.886	0.047	0.034	0.025	0.015	0.345	0.238	0.171	0.105
RandomBeacons(40,12)	79.81	74.89	71.77	68.65	0.177	0.177	0.159	0.131	1.296	1.238	1.103	0.886	0.046	0.033	0.024	0.015	0.339	0.232	0.165	0.099
RandomBeacons(40,13)	80.21	75.27	72.15	68.79	0.176	0.177	0.159	0.131	1.290	1.240	1.107	0.887	0.046	0.032	0.023	0.014	0.335	0.227	0.159	0.093
RandomBeacons(40,14)	80.48	75.64	72.48	69.01	0.175	0.177	0.160	0.131	1.284	1.242	1.112	0.889	0.045	0.032	0.022	0.013	0.332	0.223	0.155	0.089
RandomBeacons(40,15)	80.64	75.94	72.78	69.19	0.175	0.178	0.161	0.132	1.279	1.243	1.116	0.890	0.045	0.031	0.022	0.013	0.329	0.220	0.152	0.085
RandomBeacons(40,16)	80.76	76.24	73.12	69.33	0.174	0.178	0.161	0.132	1.276	1.243	1.118	0.891	0.045	0.031	0.021	0.012	0.328	0.217	0.148	0.082
RandomBeacons(40,17)	80.85	76.49	73.40	69.45	0.174	0.178	0.161	0.132	1.272	1.243	1.121	0.893	0.044	0.030	0.021	0.012	0.327	0.214	0.146	0.079
RandomBeacons(40,18)	80.90	76.72	73.62	69.62	0.174	0.178	0.162	0.132	1.270	1.242	1.123	0.894	0.044	0.030	0.020	0.011	0.326	0.212	0.143	0.077
RandomBeacons(40,19)	80.94	76.91	73.90	69.75	0.173	0.178	0.162	0.132	1.269	1.240	1.124	0.895	0.044	0.030	0.020	0.011	0.325	0.210	0.141	0.074
RandomBeacons(40,20)	80.96	77.11	74.18	69.94	0.173	0.177	0.162	0.133	1.269	1.238	1.125	0.897	0.044	0.030	0.020	0.011	0.325	0.209	0.139	0.072
TrailMax(1)	99.11	98.72	99.04	98.20	0.516	0.495	0.500	0.495	15.733	15.809	16.812	16.855							
TrailMax(2)	98.96	98.59	98.97	98.13	0.536	0.508	0.509	0.499	16.378	16.233	17.113	16.985	0.280	0.261	0.259	0.252	8.535	8.338	8.712	8.562
TrailMax(3)	98.84	98.47	98.87	98.08	0.556	0.521	0.517	0.503	17.037	16.654	17.414	17.116	0.202	0.183	0.179	0.170	6.157	5.856	6.018	5.799
TrailMax(4)	98.70	98.36	98.80	98.01	0.575	0.533	0.526	0.507	17.630	17.075	17.710	17.246	0.163	0.144	0.139	0.130	4.978	4.621	4.673	4.419
TrailMax(5)	98.58	98.24	98.72	97.96	0.595	0.546	0.535	0.511	18.295	17.478	18.016	17.385	0.141	0.121	0.115	0.105	4.287	3.884	3.870	3.590
TrailMax(6)	98.49	98.14	98.67	97.91	0.614	0.559	0.544	0.514	18.888	17.901	18.311	17.507	0.126	0.106	0.099	0.089	3.834	3.396	3.333	3.037
TrailMax(7)	98.36	98.04	98.58	97.87	0.632	0.570	0.552	0.518	19.468	18.275	18.586	17.634	0.116	0.095	0.088	0.078	3.517	3.052	2.954	2.644
TrailMax(8)	98.23	97.94	98.55	97.84	0.652	0.582	0.560	0.521	20.123	18.673	18.887	17.746	0.108	0.087	0.079	0.069	3.289	2.797	2.670	2.349
TrailMax(9)	98.11	97.84	98.49	97.80	0.672	0.596	0.569	0.525	20.745	19.132	19.188	17.880	0.103	0.081	0.073	0.062	3.116	2.600	2.449	2.119
TrailMax(10)	98.03	97.78	98.46	97.77	0.689	0.607	0.577	0.529	21.298	19.500	19.437	17.993	0.098	0.076	0.067	0.057	2.985	2.445	2.274	1.935
TrailMax(11)	97.95	97.71	98.41	97.72	0.706	0.619	0.586	0.532	21.855	19.869	19.753	18.117	0.095	0.072	0.063	0.052	2.882	2.319	2.132	1.785
TrailMax(12)	97.86	97.66	98.37	97.68	0.724	0.632	0.595	0.536	22.426	20.297	20.069	18.254	0.092	0.069	0.060	0.049	2.804	2.217	2.015	1.661

TrailMax(13)	97.78	97.61	98.34	97.65	0.743	0.644	0.602	0.540	23.053	20.689	20.281	18.369	0.091	0.067	0.057	0.046	2.745	2.132	1.916	1.556
TrailMax(14)	97.70	97.55	98.29	97.60	0.763	0.654	0.608	0.543	23.727	21.016	20.514	18.483	0.089	0.064	0.054	0.043	2.702	2.062	1.832	1.466
TrailMax(15)	97.62	97.50	98.26	97.58	0.786	0.663	0.618	0.547	24.450	21.339	20.840	18.590	0.088	0.063	0.052	0.041	2.671	2.003	1.761	1.388
TrailMax(16)	97.56	97.45	98.22	97.55	0.806	0.674	0.628	0.550	25.114	21.700	21.185	18.710	0.087	0.061	0.050	0.039	2.650	1.954	1.699	1.320
TrailMax(17)	97.51	97.39	98.20	97.52	0.823	0.685	0.637	0.553	25.636	22.075	21.478	18.823	0.087	0.060	0.049	0.037	2.635	1.912	1.644	1.260
TrailMax(18)	97.47	97.34	98.17	97.49	0.836	0.698	0.646	0.558	26.047	22.496	21.784	18.978	0.086	0.059	0.047	0.035	2.624	1.878	1.596	1.206
TrailMax(19)	97.44	97.29	98.14	97.45	0.845	0.710	0.651	0.562	26.325	22.895	21.955	19.103	0.086	0.058	0.046	0.034	2.616	1.850	1.554	1.159
TrailMax(20)	97.41	97.23	98.11	97.42	0.852	0.722	0.657	0.564	26.556	23.308	22.136	19.172	0.086	0.057	0.045	0.033	2.610	1.827	1.516	1.116
DATrailMax(1,10)	98.83	98.21	98.15	96.42	0.539	0.339	0.206	0.074	14.110	8.918	5.528	1.636								
DATrailMax(2,10)	98.70	98.13	98.18	96.57	0.553	0.343	0.208	0.074	14.489	9.030	5.551	1.640	0.291	0.178	0.107	0.038	7.609	4.687	2.860	0.836
DATrailMax(3,10)	98.59	98.03	98.17	96.62	0.569	0.348	0.210	0.074	14.890	9.136	5.596	1.644	0.210	0.125	0.074	0.026	5.478	3.292	1.984	0.571
DATrailMax(4,10)	98.46	97.91	98.11	96.65	0.580	0.352	0.210	0.075	15.163	9.226	5.603	1.645	0.170	0.099	0.058	0.020	4.422	2.601	1.544	0.439
DATrailMax(5,10)	98.34	97.80	98.08	96.67	0.603	0.360	0.214	0.075	15.803	9.455	5.712	1.659	0.147	0.083	0.048	0.016	3.813	2.191	1.287	0.359
DATrailMax(6,10)	98.25	97.70	98.04	96.66	0.608	0.361	0.214	0.075	15.917	9.446	5.695	1.657	0.132	0.073	0.042	0.014	3.417	1.924	1.115	0.308
DATrailMax(7,10)	98.13	97.60	97.98	96.65	0.619	0.363	0.215	0.075	16.180	9.495	5.709	1.656	0.122	0.066	0.037	0.012	3.149	1.741	0.999	0.272
DATrailMax(8,10)	98.03	97.53	97.95	96.66	0.642	0.372	0.218	0.076	16.839	9.744	5.803	1.668	0.115	0.062	0.034	0.011	2.965	1.616	0.915	0.246
DATrailMax(9,10)	97.95	97.45	97.89	96.66	0.675	0.386	0.224	0.077	17.771	10.179	5.977	1.706	0.110	0.058	0.032	0.010	2.836	1.523	0.852	0.227
DATrailMax(10,10)	97.88	97.38	97.85	96.63	0.707	0.402	0.230	0.078	18.688	10.627	6.177	1.729	0.106	0.055	0.030	0.009	2.747	1.452	0.804	0.210
DATrailMax(11,10)	97.83	97.33	97.81	96.62	0.711	0.403	0.232	0.078	18.806	10.645	6.222	1.733	0.104	0.053	0.029	0.009	2.685	1.396	0.767	0.198
DATrailMax(12,10)	97.78	97.27	97.77	96.59	0.706	0.398	0.230	0.078	18.617	10.486	6.146	1.721	0.102	0.052	0.028	0.008	2.641	1.353	0.737	0.188
DATrailMax(13,10)	97.74	97.22	97.74	96.57	0.701	0.393	0.227	0.077	18.453	10.336	6.053	1.707	0.101	0.050	0.027	0.008	2.610	1.318	0.711	0.180
DATrailMax(14,10)	97.71	97.19	97.70	96.55	0.698	0.390	0.225	0.077	18.352	10.222	5.981	1.694	0.100	0.049	0.026	0.008	2.589	1.291	0.689	0.173
DATrailMax(15,10)	97.69	97.15	97.67	96.53	0.696	0.387	0.223	0.077	18.295	10.124	5.917	1.684	0.100	0.049	0.025	0.007	2.576	1.269	0.671	0.167
DATrailMax(16,10)	97.68	97.12	97.65	96.51	0.696	0.385	0.222	0.077	18.269	10.048	5.867	1.683	0.100	0.048	0.025	0.007	2.569	1.251	0.655	0.162
DATrailMax(17,10)	97.66	97.10	97.62	96.47	0.695	0.383	0.221	0.076	18.241	9.977	5.835	1.673	0.099	0.047	0.024	0.007	2.563	1.238	0.643	0.157
DATrailMax(18,10)	97.65	97.07	97.60	96.44	0.695	0.381	0.220	0.076	18.225	9.916	5.795	1.669	0.099	0.047	0.024	0.007	2.560	1.226	0.631	0.153
DATrailMax(19,10)	97.64	97.05	97.56	96.41	0.694	0.379	0.219	0.076	18.215	9.862	5.778	1.664	0.099	0.047	0.023	0.007	2.557	1.217	0.622	0.150
DATrailMax(20,10)	97.64	97.03	97.53	96.39	0.694	0.379	0.219	0.076	18.202	9.829	5.745	1.655	0.099	0.046	0.023	0.006	2.555	1.210	0.614	0.146

Table B.1: Results for the suboptimality experiment for all four sets of maps.