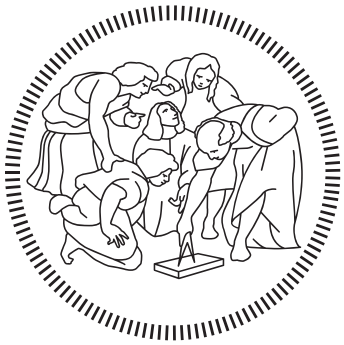
POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria
Master of Science in Computer Science and Engineering



Design and implementation of an optimal algorithm for solving visibility-based pursuit evasion games

Supervisor: Prof. Francesco AMIGONI
Co-supervisor: Dr. Alberto QUATTRINI LI

Master Thesis by:
Raffaele FIORATTO ID 854261

Academic Year 2016 - 2017

Raffaele Fioratto: *Design and implementation of an optimal algorithm for solving visibility-based pursuit evasion games* | Master of Science in Computer Science and Engineering, Politecnico di Milano.
© Copyright December 2017.

Politecnico di Milano:
www.polimi.it

Scuola di Ingegneria Industriale e dell'Informazione:
www.ingindinf.polimi.it

Ringraziamenti

Sono passati ormai cinque anni da quando ho iniziato la mia carriera da studente al Politecnico e ringraziare in una pagina tutte le persone che ho incontrato e mi hanno sostenuto in questo mio percorso è praticamente impossibile.

Desidero ringraziare il Prof. Francesco Amigoni e il Dott. Alberto Quattrini Li, per il loro fondamentale aiuto nella elaborazione di questa tesi e per la loro disponibilità a chiarire ogni mio dubbio.

Un grazie va alla mia famiglia, i miei genitori e alla mia sorellina per avermi sempre sostenuto (e sopportato) in tutti questi anni. Un rigraziamento anche ai miei nonni, in particolar modo a mio nonno Natale, che, indipendentemente da dove tu sia in questo momento, so che sei fiero di me. Ci manchi moltissimo. Un grazie particolare va anche a Cinzia, per la sua simpatia, il suo affetto, e per tutto il supporto che mi ha dato e, sono sicuro, continuerà a dare. Un ringraziamento va anche a Roberto.

Un grazie speciale va a tutti i miei colleghi e compagni di Università con cui ho condiviso tante ore in aula, ma non solo, e che mi hanno fatto crescere anche come persona. Un rigraziamento particolare va a Nicolò, per tutto il tempo che abbiamo passato insieme sui progetti. Alla fine tutta la fatica spesa ha dato i suoi frutti e spero che ciò valga anche per te.

Infine un rigraziamento particolare va a te Federica, che in tutti questi anni mi hai sopportato e continui a farlo, anche se non so come tu ci stia riuscendo. Grazie specialmente in questo ultimo periodo, in cui abbiamo fatto tanti sacrifici e, nonostante tutto, tu abbia sempre continuato a credere in me. Questo è un altro piccolo ma grande passo verso la realizzazione dei nostri obiettivi.

Milano, December 2017 R. F.

A Federica...

Contents

1 Introduction **1**

2 State of the Art **5**
2.1 Adversarial pursuit-evasion games . . . 6
2.1.1 Pursuit-evasion games in geometrical environments . . . 7
2.1.2 Pursuit-evasion games on graphs . . . 10
2.2 Probabilistic pursuit-evasion games . . . 12
2.3 Differential games . . . 13
2.4 Minimax algorithm . . . 14
2.5 Monte Carlo Tree Search . . . 17
2.6 Computational geometry . . . 18
2.7 Wrap up . . . 19

3 Pursuit-evasion in simple polygons **23**
3.1 Formal definition . . . 23
3.2 Research hypotheses . . . 25
3.2.1 Main contributions of this thesis . . . 25
3.3 The case study . . . 26
3.3.1 Indoor maps as simple polygons . . . 26
3.3.2 Rash evader setting . . . 26

4 Solving the problem **29**
4.1 Algorithm overview . . . 29
4.1.1 Splitting the algorithm in two phases . . . 29
4.1.2 Partitioning the environment . . . 30
4.2 Phase 1: searching the evader . . . 32
4.2.1 Search as a graph covering problem . . . 32
4.2.2 Constructing a node . . . 35
4.2.3 Updating the frontier . . . 36
4.2.4 Complexity of the algorithm . . . 37
4.3 Phase 2: capturing the evader – Minimax . . . 37
4.3.1 Setting up the search problem . . . 38
4.3.2 Finding a solution and speed ups . . . 39
4.4 Phase 2: capturing the evader – Monte Carlo Tree Search . . . 41
4.4.1 Using random sampling when making decisions . . . 42
4.4.2 The exploration vs. exploitation dilemma . . . 45

5 Implementation **47**
5.1 The simulation software . . . 47
5.1.1 The core module . . . 48
5.1.2 GUI module . . . 48
5.1.3 CLI module . . . 50
5.1.4 Creating the environments . . . 50
5.2 Frameworks and technologies employed . . . 51
5.2.1 Programming languages and paradigms . . . 51
5.2.2 CGAL . . . 52
5.2.3 Qt . . . 53
5.2.4 Boost . . . 54
5.3 Code tweaks . . . 54

6 Experiments and results **57**
6.1 Experimental setup . . . 57
6.1.1 Environments . . . 57
6.1.2 Parameters . . . 59
6.1.3 Batch of simulated searches . . . 61
6.1.4 Metrics used for evaluation . . . 62
6.2 Clearing the environment with different discretization points . . . 62
6.3 Minimax with different speeds . . . 66
6.4 Comparing Monte-Carlo Tree Search with Minimax . . . 68
6.4.1 First scenario: both players use Monte-Carlo Tree Search . . . 68
6.4.2 Second scenario: pursuer uses Monte-Carlo Tree Search and evader uses Minimax . . . 70

7 Conclusions and future work **73**

A System documentation **77**
A.1 Architectural structure . . . 77
A.2 Algorithms implementation . . . 78
A.3 GUI implementation . . . 87
A.4 CLI implementation . . . 88

B Additional tools developed **89**

Bibliography **101**

List of Figures

2.1 Solution to the original lion and man game 8
2.2 Omni-directional and unlimited view vs. limited FOV and range . . 9
2.3 Example of Minimax game tree 16
2.4 Outline of Monte-Carlo Tree Search as showed in [20]. 17

3.1 **(a)** An example of a simply-connected polygon map. **(b)** Even assuming the worst case position of the evader (red point), it doesn't gain any advantage using a general strategy. 27

4.1 Example of a partial computation of the distance matrix using the grid decomposition. 31
4.2 Example of visibility-based decomposition of a map. 32
4.3 Intersecting a contaminated region's edge with the visibility-based decomposition. 35
4.4 Polygonal difference between the environment and cleared region (colored in gray). 36
4.5 Neighbor cells with speed $k = \frac{\nu}{e} = 1$ **(a)** and with $k = 4$ **(b)** 40

5.1 Main screen of the graphical simulation application. 49
5.2 Example of a completed simulation with the pursuer as a winner. . 49
5.3 Example of a partial simulation using the CLI application. 50

6.1 Environment *albert-b-laser*. 58
6.2 Environment *utk-claxton*. 58
6.3 Environment *fr079*. 58
6.4 Environment *ubremen-cartesium*. 59
6.5 Cells comparison histogram. 61
6.6 Average and standard deviation of time steps needed for detecting the evader changing discretization points (see Table 6.2). 64
6.7 Average and standard deviation of computational time for detection algorithm with different discretization points (see Table 6.2). 64
6.8 Average and standard deviation of number of times grid decomposition is used in the detection algorithm as a function of discretization points (see Table 6.2). 65
6.9 Number of simulations where at least in one time step the grid decomposition is used replacing the visibility-based one. 65

6.10 Comparing two opposite settings executed in the map *albert-b-laser*. The pursuer walks along the green path, while the evader's route is the red one. 67
6.11 Plots of the values reported in Table 6.6. 67
6.12 Plots of the values reported in Table 6.7 when both players compute their strategy using MCTS. 69
6.13 Plots of the values reported in Table 6.8, when the pursuer employs MCTS while the evader uses Minimax. 71

A.1 Class diagram of the software with the main classes and their relationships 79

List of Tables

2.1 List of pursuit-evasion games. 20

6.1 Environments with their sizes. 59
6.2 Number of discretization points across different runs. 60
6.3 Number of visibility and grid cells for each map. 60
6.4 Speed of the robots across all the maps. 61
6.5 Number of points selected along with the total amount of settings $(points \cdot (points - 1))$ 62
6.6 Results with average and standard deviation over the samples for each map in both configurations of the movement speed of the robots. 67
6.7 Results with average and standard deviation over the samples for each map in both configurations regarding the relevant parameters in MCTS. 68
6.8 Results with average and standard deviation over the samples for each map in both configurations regarding the relevant parameters in MCTS. 70

Listings

B.1 *raster_to_poly.py* script. 89
B.2 *scale_poly.py* script. 99

List of Algorithms

1 Planning the motion strategy in the detection phase. 33
2 Constructing a new graph node 35
3 Updating the frontier used in the detection phase 36
4 Monte Carlo Tree Search algorithm 42
5 Playout simulation algorithm 43
6 Selecting a state using random sampling from a geometrical distribution 44

Abstract

A pursuit-evasion game is a non-cooperative game in which a pursuer tries to detect or capture an adversarial evader. We study a pursuit-evasion game which takes place in a known polygonal environment. The goal of the pursuer is to capture the evader by moving onto its location. The players can know each others' locations only if they can "see" each other – i.e., if the line segment connecting their locations lies entirely inside the polygonal environment. The complexity of representing the information available to the players at a given time makes solving pursuit-evasion games with visibility limitations difficult. We represent the state of the game using an efficient visibility-based decomposition of the environment paired with a more classical grid-based decomposition. The optimal players' strategies are computed using Minimax search improved with some speedup techniques that preserve optimality. We show that our approach is complete for a rash evader, which hides from the pursuer and does not move from its hiding location when the pursuer is not visible. Simulations in realistic indoor environments show the viability of using the proposed approach, compared to a Monte Carlo Tree Search.

Sommario

Il pursuit-evasion è un gioco non cooperativo in cui un pursuer cerca di individuare e catturare un evader. La versione del gioco di pursuit-evasion preso in considerazione in questa tesi si svolge in un ambiente poligonale noto. L'obiettivo del pursuer è di catturare l'evader muovendosi nel suo stesso punto. I giocatori possono conoscere la loro reciproca posizione solo se sono in grado di "vedersi" – cioè, se il segmento che unisce la loro ubicazione corrente è contenuto interamente all'interno dell'ambiente poligonale. La complessità necessaria a rappresentare l'informazione disponibile ai giocatori in un dato istante di tempo rende di difficile risoluzione i giochi di pursuit-evasion con visibilità limitata. In questa tesi, lo stato del gioco è rappresentato efficientemente tramite una decomposizione basata sulla visibilità, accoppiata con una più classica basata su griglia. Le strategie ottime dei giocatori sono calcolate usando una ricerca Minimax con l'aggiunta di diversi metodi che ne aumentano la velocità senza sacrificare l'ottimalità. Viene mostrato che l'approccio proposto è completo per un rash evader, il quale usa una strategia che consiste nel nascondersi dal pursuer e rimanere fermo fintantoché il pursuer non diventi visibile. Sono state eseguite diverse simulazioni in ambienti realistici per valutare la bontà del metodo proposto, includendo un confronto con un approccio basato su Monte Carlo Tree Search.

Chapter 1

Introduction

A *pursuit-evasion* game is a non-cooperative game in which a pursuer tries to detect or capture an adversarial evader. In recent years, the attention towards this problem is increasing thanks to its extreme flexibility that permits to adapt it to a plethora of different applications, such as search-and-rescue, modeling animal-behavior, flight traffic control, and so on. In a pursuit-evasion game one or more agents, called *pursuers*, try to detect or capture one or more agents, called *evaders*, which, on the other hand, try to escape indefinitely.

Given its practical importance, the problem has been studied in several contexts and with several assumptions. This work studies the pursuit-evasion game from a geometrical point-of-view with a single pursuer and a single evader. The players fully know the environment but they can see each other only if line-of-sight is established. The pursuer has to first explore the environment in order to detect the escaping evader and then plan a motion strategy that secures capture.

The aim of this thesis is to develop and analyze experimentally an *optimal* algorithm that allows the pursuer to capture an escaping evader in a geometrical environment using a search-based approach, typical of Artificial Intelligence, on top of a state-space representation of the problem. The selected algorithm is Minimax and allows to compute an optimal strategy that is able to minimize the number of moves necessary for the purser to capture the evader.

The proposed algorithm is implemented using a classical object-oriented programming language alongside with an application built from scratch that performs simulations of realistic indoor test environments modeled as simply-connected polygons. The results have been evaluated as a function of different parameters, such as the velocity at which the players move, and according to several metrics, like the computation time necessary to find a solution of an instance of the problem.

The validity of the algorithm is evaluated against an implementation of a Monte-Carlo Tree Search using a modified version of the simulation software. The comparison has been made using the same environments in order to emphasize differences and similarities in terms of performance but also regarding the quality of the solution with respect to the optimality criteria of interest, which are the number of moves necessary for catch the evader and the number of times that a region needs to be cleared.

Further research can be made in order to enhance even more the proposed technique considering more difficult and complex scenarios that ultimately may

lead to a real-world implementation of the whole approach.

A typical example found in the literature of pursuit-evasion games is called *lion-and-man* in which the objective is to find a strategy that guarantees capture of the man by a lion for every possible instance of the problem. Since the environment is geometrical, this kind of games sits at the intersection between Artificial Intelligence, Robotics, and Computational Geometry.

A pursuit-evasion game is usually composed of two distinct phases: a *detection* phase and a *capture* phase. In the detection phase the pursuer usually doesn't know the position of the evader (the opposite may not be true), and executes a strategy that clears the environment to establish *line-of-sight* with its adversary. On the other hand, the evader may employ elusive techniques with varying degrees of complexity to hide from its adversary. Anyway, if the pursuer is able to detect the evader, the next objective is to perform the actual capturing, without losing line-of-sight, condition that would regress the game to the previous phase.

In order to precisely define the start and end of these phases a characterization of the environment and the agents is made. One of the main aspects of a pursuit-evasion game is the description of the capabilities of the players depending on the possibilities allowed by the environment. Typically, it is necessary to describe the aspects concerning the motion and sensing capabilities. It is not mandatory that the agents have the same abilities, as [43, 61] show, they can also have different velocities, like for instance a fixed bounded speed for the pursuer, and an infinite velocity for the evader. Regarding the environment, the problem has been studied in two distinct ways, namely *geometrical* and *graph-based*. Historically research has been done first on the former setting, while the latter came a couple of decades later. The already introduced lion-and-man game falls in the first category and appears in [57]. The game is studied in several ways, like extending the number of the players, which often depends on the characteristics of the environment [50], or focusing on generalizing as much as possible the problem guaranteeing strategies that work on a wider class of geometrical entities and not only on polygons [3]. A different paradigm encompasses the utilization of graphs to model the environment. In contrast to the geometrical setting, in which the environment is defined as a continuous entity, here a discretization process is applied. Nodes typically represent the positions of points of interests, such as rooms, and edges establish a link among them representing a doorway or a corridor that connect one or more rooms; in the literature the seminal work about this paradigm is by Parsons [65]. This thesis uses a geometrical representation of the environment.

The original lion-and-man formulation assumes that players have full knowledge not only about the environment but also on the position of their adversary in every situation. This doesn't apply in the thesis, where the concept of limited visibility is used. The idea is introduced for the first time by Suzuki and Yamashita [77] and builds upon the concept of *visibility region* of a player, which is the set of points that can be joined from the position of such player using a line segment that lies entirely inside the environment's boundary. In order to explore the map and get knowledge on the adversary, the players are usually equipped with one or more sensors that are able to perceive the surroundings. Some works concentrated on modeling such sensors as *omni-directional* with unlimited *range* [28, 69, 76], which

is also the case in this thesis. Other works considered more realistic scenarios in which the sensors have limited range and field of view [15, 42].

The approach utilized in this thesis is derived from [69] which applies a search-based paradigm using classical adversarial informed search algorithms such as Minimax to solve the problem. The players have the same speed and are able to move inside the environment in a turn-based fashion, with the pursuer moving first. Being Minimax a search-based algorithm, the problem has to be modeled using a state-space representation. The first issue that arises is a proper choice regarding the knowledge representation in the states. A classical approach is to encode the position of the players using the center of the cells derived from a regular grid decomposition of the environment. The critical aspect is the size of the edge: if it is too small, the state-space becomes too fine grained, compromising the tractability of the problem, if it is too coarse grained, the information encoded in the representation becomes inaccurate. In order to overcome this problem, a different method for decomposing the environment is employed in this thesis, consisting of connecting mutually visible vertices of the polygon with a line. The intersections of these lines induce the generation of a number of cells that is significantly lower than in a classical grid decomposition. The strategy employed by the evader is also peculiar: it holds its position until line-of-sight is established with the pursuer; the strategy is called *rash evader* and allows to solve efficiently the game. Compressing the state-space is usually not enough: the downside of using Minimax is its exponential nature with respect to the worst case number of moves that are possible starting from a state of the game, and the number of moves necessary to reach a *terminal state*, where the game ends. This thesis addresses the issue by building upon this algorithm several enhancements that dramatically increase the performance without sacrificing optimality. In the literature, the concern about optimality of pursuing strategies is not usually addressed. As a matter of fact, the number of works that seek for optimal approaches for solving pursuit-evasion games in geometrical environments is scarce, which is why this thesis stresses about this topic.

The proposed algorithm is evaluated using nearly 10,000 simulations with different parameters in several test environments, first by employing a low speed for both players, that is subsequently increased. A comparison between these two settings is made emphasizing on the differences and similarities.

This comparison is further expanded by implementing a Monte-Carlo Tree Search algorithm that serves as a benchmark for the proposed Minimax implementation. The aim of this comparison is to assess the performance and the robustness of the proposed algorithm against a standard approach for solving difficult search problems.

The produced results for all the comparisons are presented in a dual representation, namely tabular and on plots, to better visualize the key points that characterize each method also as a function of the most important parameters. Results show that the proposed approach outperforms the one introduced in [69] on the same environments and it is robust against new maps as well. Compared to a Monte-Carlo Tree Search implementation which, in general, generates sub-optimal strategies, the proposed algorithm computes an optimal motion plan also in lower computational time with respect to MCTS.

Outline

The thesis has the following structure:

In Chapter 2 the state of the art about pursuit-evasion is shown, focusing in particular on the works that are closely related with this thesis. The theory behind the methods employed in the proposed algorithm is also presented; being them general and well-suited also for completely unrelated topics, a link with the pursuit-evasion context is established focusing on the works that have already tried to put together all these concepts.

In Chapter 3 there is a formal definition of the problem being analyzed, alongside with a set of research motivations that are the cornerstones for the work that this thesis illustrates. Furthermore, an explanation about the context is given, characterizing the class of environments considered and the general ideas behind the strategy that the players employ.

In Chapter 4 the proposed method is illustrated along with the set of enhancements that improve its performance without sacrificing optimality. Insights about Monte-Carlo Tree Search, which is utilized for the comparison made in Chapter 6 are also given, showing that most of the implementation decisions can be used again, changing only the relevant modules.

In Chapter 5 a practical implementation of an application that simulates the game is presented. The main modules that compose the software are identified and explained together with the relevant third-party frameworks and code tweaks that work in close synergy with the formal speedup methods presented in Chapter 4.

In Chapter 6 the implemented solution shown in Chapter 5 is thoroughly evaluated. The results of such experimental evaluation are reported considering also different choices of the most important parameters in realistic indoor environments. The chapter compares Minimax and Monte-Carlo Tree Search.

In Chapter 7 the conclusions about the work are discussed alongside with several future studies that arise from this thesis.

In Appendix A the architecture of the system that implements the proposed algorithm is documented and explained. The interaction of the modules introduced in Chapter 5 is precisely illustrated down to the actual piece of code that carries out a specific task.

In Appendix B some companion Python scripts are presented that helped the creation of the maps selected for the experimental evaluation. Starting from simple raster image of a map, these tools generate a representation that is complying to the format expected by the simulation software.

Chapter 2

State of the Art

This chapter describes the state of the art regarding pursuit-evasion games in different settings, in particular, focusing on adversarial games that take place in a polygonal environment, which is the configuration that this work considers. The main objective of this chapter is to provide to the reader all the necessary insights in order to achieve a sufficient knowledge of the research field and to understand the rest of this thesis. If the reader is interested and wishes to study deeper this topic, the survey by Isler, Hollinger, and Chung [36] is a good starting point.

The problem of pursuit-evasion has been studied through decades adopting different settings and hypotheses, up to the point that is hard to perform an exhaustive classification. The most natural way to define such problems is to consider an adversarial setting in which there are two players, one is trying to capture another, which wants to escape. Section 2.1 address this kind of games. Additionally, the problem can be further divided by considering the kind of environment in which it takes place, namely geometric or graph-based: the former is more realistic while the latter is more abstract but still important from a theoretical point of view. These scenarios are further explained in dedicates sections (Section 2.1.1 and Section 2.1.2).

A different paradigm that can be used to study pursuit-evasion is to characterize the behavior of the players using a probabilistic model that in general is independent for each of them. With this approach there is no such concept of adversaries, but it is still relevant to develop a strategy such that the position of both players is the same given an instant of time. This kind of problems is illustrated in Section 2.2.

A third approach is to use a paradigm similar to the one used to solve control problem by using physical quantities and laws. This class of problems is called differential games and are explained in Section 2.3.

As an important notice, in this brief explanation I voluntarily defined the problem using a single pursuer and evader for sake of clarity, but as the rest of this chapter explains, studies address the problem employing a higher number of pursuers, or evaders, and, in some cases, both of them at the same time.

Section 2.4 and 2.5 deal with Minimax and Monte-Carlo Tree Search algorithms respectively, which are the two basic blocks used for the algorithm that is the main contribution of this thesis and it is described in later chapters.

Since this study approaches the problem in a geometrical setting, Section 2.6 shows the most important results that help solving related problems from the

field of computational geometry, such as computing set operations on polygons, or finding out the intersection across geometric entities in an efficient way. Some of the insights coming from such work are used to develop the proposed algorithm in this thesis.

2.1 Adversarial pursuit-evasion games

One of the peculiarities of classical Artificial Intelligence is that its foundations are comprised of elegant and formal mathematical models that are able to suit for a broad class of problems that, at first sight, seem unrelated. Fortunately, adversarial pursuit-evasion games are embraced by these kind of models. In particular, the players can be seen as *agents* which, according to Russell and Norvig [71], are able to sense the surrounding environment thanks to the help of sensors and make actions using actuators. Moreover, they are *rational*, i.e., they try to maximize some performance measures given what they know.

The problem of motion planning in such adversarial settings opens up a wide range of applications in real world scenarios. Classical cases are catching burglars, or playing hide-and-seek, but pursuit-evasion can be applied also in search-and-rescue context and, beyond robotics, to flight traffic control, to model animal behavior, and even to intercept hostiles missiles and to network security.

The number of variables that one needs to take into account when designing a pursuit-evasion game, and the combination of them results in a virtually infinite number of instances and settings. The rest of this section shows the most important characteristics that can be extrapolated in order to describe in an unified and coherent way as much pursuit-evasion games as possible.

Environment. This is probably the first thing that comes to the mind of the reader when speaking about pursuit-evasion. It is possible to distinguish two kinds of environments: geometrical and graph-based. The former setting is also called *continuous* because the environment can be defined as the set of infinite points that are delimited by a boundary. In the simplest case, it can be two-dimensional, namely a polygon, but also three-dimensional. Historically, the two-dimensional case has been studied more thoroughly compared to three-dimensional one because it is in general simpler to model but still valuable in real world scenarios. It is important to notice that, usually, some sort of discretization of such polygons must be taken into place. The most common way is to partition the environment using a grid but other techniques can be employed. Modeling of the environment as a graph is different, with edges and nodes used to describe it. In particular, players sits on nodes and can move along edges. Nodes encode the positions, namely a room, inside the environment, while the edges establish a link among them.

Information available to players. An important point is to define what players *know* about the game. In particular, if they already have complete knowledge about the environment, the position of other players, if they are allowed to discover at some point the strategy of their adversaries, and so on. A small variation of

these conditions can dramatically change the complexity of the problem and its applicability in real contexts.

Capabilities of the players. Expressing the capabilities in terms of movement and sensing for the players is also critical. For instance both pursuer and evader can move arbitrarily fast, or having bounded speed, their sensors can have limited range, and so on. In some settings, one player can have an advantage: for instance the evader can move as fast as it wishes, whereas the pursuer must move at a fixed speed.

Capture condition. Sometimes one is interested only in maintaining visibility instead of capturing the evader. In certain settings, it is enough that the distance between the pursuer and the evader is within a predetermined range in order to trigger the capture condition instead of the agents being in the same position.

Number of players. It might be interesting to study if a group of pursuers can eventually capture a single evader or if multiple evaders can be captured by a single pursuer. The number of players in one team could be known by the other one or not.

Time domain. The most used choice is discrete time but also continuous settings are allowed.

Order of play. Players can move when it is their turn or at the same time.

As the reader can see, the problem can be defined and studied in different ways. The next section shows the major studies that are more similar to the one discussed in this thesis, and what are the similarities but also the differences that motivate this work.

2.1.1 Pursuit-evasion games in geometrical environments

The seminal work about this kind of formulation is done by Littlewood [57], which calls it *lion-and-man*, a name that it is still used nowadays. The original problem states that a lion wants to catch a man in a circular environment, the game is played in turns and they make moves alternately. Besides the formulation, the author also shows a *complete* strategy, which ensures that eventually will terminate successfully for every possible initialization of the problem, for the lion: if it moves along the segment that connects the center of the map and the position of the prey, the lion is able to reduce its distance to the target and eventually catches the man (Figure 2.1). This work is extended in [72], where almost optimal bounds for capturing the man are given. Throughout the decades, the main focus, like in [11, 28, 53], is to find at least a complete algorithm for a particular setting that is often introduced for the first time in the work itself, while in this thesis the focus is shifted towards not only a complete but also *optimal* algorithm with respect to an objective function depending on a specific application.

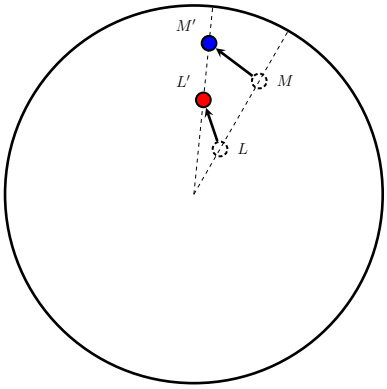


Figure 2.1: Solution to the original lion and man game

Using the term "geometrical" is often not enough. A lot of studies characterize the environment more precisely, for instance they consider *simply-connected* polygons, in the simplest form without obstacles or holes like in the original lion-and-man game. A polygon is simply-connected if its boundary is composed of straight, non-intersecting line segments. If we consider holes, the scenario becomes much more realistic and therefore a lot of work focuses in this direction. In particular, regarding holes, there is the natural extension of the original lion-and-man game [11] which also shows that even with an arbitrary number of holes, three pursuers are sufficient to capture one evader. A similar condition has been found in higher dimensional environments [50] where at least $d + 1$ pursuers are needed in $\mathbb{R}^d$, focusing on the particular case of three dimensions [45], in which three pursuers are sufficient and sometimes necessary, [10] extends the result also in topological spaces. Alexander, Bishop, and Ghrist [3] study the setting considering *Convex Euclidean Domains* and in environments exposing the *CAT(0) property* [4], again all of these works focusing only completeness rather than optimality. A CAT(0) space is a *metric space*, which is a set of elements called points where the concept of distance is defined, in which the sum of the angles of each *geodesic triangle* is no greater than π. Convex Euclidean Domains are an example that satisfy such property. A geodesic triangle is composed of three different points that are pairwise-connected with a *geodesic line* which is the shortest curve in a space (in a broader sense, even in higher dimensions) that connects such points.

Bhadauria et al. [12] develop two slightly different methods based on strategies to trap the evader employing exactly three pursuers: in the Minimal Path Strategy a *visibility graph*, whose nodes represent the vertices of the polygon, and edges link pairs of nodes that have line-of-sight visibility, is constructed and the first three shortest paths are selected. In the Shortest Path Strategy, the shortest path is chosen in the evader's region. They find out that not only three pursuers are sufficient, as mentioned earlier, but also the number of moves necessary are upper bounded by $O(n \cdot diam(P)^2)$ with P being the polygon having n vertices, and $diam(P)$ being its diameter, i.e., the longest distance between a pair of vertices. Recently, Ames et al. [6] manage to tighten the bounds: if pursuers coordinate themselves using a leapfrog strategy, consisting of alternatively guarding/increasing the cleared area, it is demonstrated that two of them are sufficient to catch an evader that has the same speed in any polygonal environment P with n vertices

Figure 2.2: Two different types of visibility polygon on the left pursuer (the red dot) is equipped with an omni-directional sensor that has unlimited range and it is able to detect the evader (the blue dot), whereas, on the right, the same situation but with a limited field of view and range that is not sufficient to spot the evader.

and h holes, computing an upper bound of $O(n \cdot h \cdot diam(P))$ moves. The main assumption is that all the players have global visibility.

Up to now we consider the original formulation, in which *global visibility* of both players is assumed, i.e., they can see everything that is happening in the environment and therefore they know the position of each other during the whole game. Researchers are interested in enriching this model in order to apply it directly in robotic scenarios, therefore, more realistic assumptions and hypotheses have been developed. In particular, the *visibility-based* pursuit-evasion, where the players (or only some of them) are equipped with some sensors that are able to perceive the surroundings and make decision based on what they see in their visibility polygon. Suzuki and Yamashita [77] first introduced this concepts.

Guibas et al. [28] develop a complete algorithm for players using perfect omni-directional sensors with unlimited range (Figure 2.2 **(a)**), they also determine that $O(\log n)$ pursuers are sufficient for a simply connected polygon with n vertices without obstacles, at most $O(h + \log n)$ pursuers within a polygon having h holes and at least $\Omega(\sqrt{h} + \log n)$, though they are interested only in locating the evader. An extension of their work [74], proves that a complete algorithm exists also for capturing. A related work, assuming bounded maximum speed for both players, has been done by Tovar and LaValle [79], proving the existence of a complete algorithm. In this setting, some efforts are made also in finding a algorithm for planning an optimal motion strategy. In particular, [75, 76] focus on locating a moving evader with a single pursuer, by ensuring that previously cleared shadows are not recontaminated again. By using a branch and bound algorithm and some pruning techniques, the computational cost can be dramatically decreased at cost of losing optimality. Regarding an optimal strategy for capturing, Quattrini Li, Amigoni, and Isler [69] apply the paradigm of informed search algorithms through minimax using a novel method for compressing the state space and some speedup techniques for pruning sub-optimal branches. Still the combination of high depth and branching factor in certain maps and instances might result in the impossibility

to find a solution in reasonable time.

A more realistic scenario involves limiting the capability of sensing the environment, for instance, [15] study the lion-and-man game in which the lion can observe its prey's location only when it is within a suitable range, proving that exists a strategy that is sufficient for capturing the man inside bounded convex planar environments. Karnad and Isler [42] study a small variation of the problem that takes place in the first quadrant where the pursuer can sense the environment and takes measures, resulting in a strategy that guarantees capturing in a bounded number of steps. Moreover, this study is extended to free space connected polygons yielding, once again, a complete strategy for a single pursuer [26] using also sensors with limited FOV like in Figure 2.2 **(b)**.

Sometimes detecting the evaders neglecting capturing is enough. LaValle et al. [53] (which Guibas et al. [28] extend), study the visibility-based setting in which the evader is capable of moving arbitrarily fast, proving that a single pursuer is sufficient for a complete detection algorithm, a similar result has been obtained even if the evader is way more powerful [37] by employing a randomized strategy, capturing condition was also studied yielding that two pursuers which at least one fast as the evader, are sufficient for capturing in a simply-connected polygon. Polygons with obstacles have been also considered [13, 59] in which players have omni-directional sensors and limited range. They show that by constructing a suitable graph, pursuers can capture the evader though proving such problem is NP-complete. Other examples are [64] and [54] which extend the problem in three dimensions resulting in a complete algorithm for determining a motion strategy to capture the evader.

Finally there are peculiar cases that do not fall in the previous categories but are still worth mentioning. Katsev et al. [43] develop a strategy for both mapping and detecting one or more pursuers under the constraint of moving along the walls. They manage to find out an algorithm but its completeness is still an open research subject. Durham, Franchi, and Bullo [24] develop a distributed algorithm under realistic assumptions: pursuers have limited sensing and communication capabilities along with a finite memory, also they don't know the map nor the position neither trajectories of the pursuers, still they are able to find out a complete algorithm with reasonable performance compared with a Monte Carlo alternative.

As a last example, Noori and Isler [60] study the problem in a particular class of polygons called X-monotone, despite the evader having global visibility and knowing the strategy of its adversary whereas the pursuer does not, still, a complete algorithm exists with worst case time complexity of $O(D^{13} \cdot n^7)$ with D being the diameter and n the number of vertices.

As the reader can notice, almost every work regarding pursuit evasion in a geometrical environment focuses in developing an algorithm that only satisfies the completeness condition. This thesis tries to make a major step forward focusing not only on completeness but also, and most importantly, on optimality.

2.1.2 Pursuit-evasion games on graphs

This formulation appears for the first time in [65], which depicts a man lost in a dark cave, where he can move arbitrarily fast, teleport to nodes, and know all the

moves made by the searchers. The paper defines the number of searchers necessary to catch the evader and an algorithm to compute such number for arbitrary finite connected graphs.

Nowakowski and Winkler [61] introduce another formulation called *cops and robbers*, in which a single pursuer can be arbitrarily faster than evaders, all with global visibility. They characterize the structure of graphs that would allow a single cop to win. Later on, [27] analyze time complexity using different configurations: employing k cops in an undirected graph and a single robber the problem is EXPTIME-complete. Such class of computational complexity contains all decision problems that have exponential run time. Completeness means that if there exists a way to reduce the complexity of such decision problem to polynomial, every other ones that belong to EXPTIME are reduced as well: this implies that EXPTIME-complete contains the hardest problems in EXPTIME. The same result can be applied with k robbers in a strongly connected graph in which each player can choose its initial position. A slightly different formulation appears in [2] in which robbers are *passive*, i.e., they are allowed to stand still when it is their turn, whereas in the previous work they are obliged to move at every turn. In this configuration, three cops are sufficient to win in planar graphs. This behavior is similar to the rash evader strategy employed in this thesis. In the passive robbers case, though, there is nothing that prevents the robbers to move even if they are not visible to the cops, whereas an evader playing the rash strategy must not move as long as there isn't line-of-sight with the pursuer.

Just like in geometrical setting, the limited visibility case is studied, in particular there is the *hunter-and-rabbit* game, which is played by a single pursuer and evader that can detect each other only if they sit in the same vertex. With this formulation, [1] manage to develop a hunter strategy for general graphs with $O(n \cdot \log(diam(G)))$ moves against *restricted* rabbits, that are allowed to move along edges and *unrestricted* rabbits that can jump to an arbitrary node. Moreover, their strategy for unrestricted rabbits is optimal up to constant factors. Here, $diam(G)$ denotes the diameter of graph G, which is the greatest distance between any pair of vertices. Isler, Kannan, and Khanna [38] allow multiple hunters to catch a single rabbit, again having limited visibility, proving that there exists a polynomial time algorithm that is able to decide whether, given a graph, it is possible for the hunters to catch their prey.

Isler and Karnad [39] study a mixed approach with respect to visibility, starting from the original cops and robbers game employing a single pursuer, in which the cop has limited visibility, and the ability of seeing nodes at distance at most k, whereas the robber has global visibility. With general graphs, the cop is able to detect the robber, and on a particular class of graphs, a greedy strategy is sufficient to guarantee the capture of the evader.

Kolling and Carpin [48] introduce the GRAPH-CLEAR problem: given a weighted graph $G = (V, E, w)$ composed of contaminated edges E and vertices V, the aim is to plan a strategy S to clear G minimizing the cost. Besides the formulation, the authors study the problem restricted to trees, which are graphs without cycles, developing a complete algorithm. Subsequently, [47] show that the GRAPH-CLEAR problem is solvable in linear complexity on trees but is NP-hard in general, moreover the authors develop an algorithm applicable to trees that allocates

the minimum number of pursuers needed to catch multiple evaders and has $O(n^2)$ time complexity where n is the number of vertices. With spanning trees, multiple pursuers, and one evader the upper bound can be tighten to $O(|N|\log|N|)$ [30], being $|N|$ the number of nodes.

Britnell and Wildon [16] study another variation of the original cops and robbers game, in which the cop (called the prince) has no knowledge about the position of the robber (the princess). Nevertheless they manage to determine the class of graphs in which the pursuer can capture the evader with also the minimum number of steps.

Recently, an approach based on game theory has also been studied by Konstantinidis and Kehagias [49], for a variant of the original problem: the cops are *selfish* meaning that one of them winning, results in a *partial* loss for the others, of course meaning a complete loss for the robber. The main result is to prove the existence of Nash equilibria in deterministic strategies in these setting.

Amigoni and Basilico [7] spend some effort into finding an optimal strategy: the environment is modeled on a bounded 8-connected grid that has m entrance cells and n exit cells. The game consists of a single evader that enters and exits the grid map and a pursuer that tries to intercept the adversary along its path. Constrained optimization is utilized to solve the game. Besides the different assumption made for the environment, the criteria that optimality adopts is the probability of capturing an evader, which is not the same considered in this thesis.

At last, in [32] the search is treated as a resource allocation problem, resulting in a scalable algorithm that is able to clear the environment utilizing multiple pursuers that work in a worst-case adversarial scenario and has also good average performance in a non-adversarial context.

Since this kind of problems make different assumptions about the environment, it is not possible to make a direct comparison to the one studied in this thesis.

2.2 Probabilistic pursuit-evasion games

Robotics has to deal with real world, and when it comes to design and develop solutions that have to be applied to real-world situations, you have take into account uncertainty. For instance, when making a move, a robot can fail to cover the distance that was planned for a plethora of different causes, the floor can be slippery, the power source failed to deliver the correct current to the motors or another failure occurred. Also sensors are not perfect, the noise that affects the acquired data always leads to some uncertainty when reasoning about what do next.

That's when probability comes in to place: such uncertainty can be modeled using some kind of probabilistic model. This approach is also utilized when characterizing the location of the target or its motion, with the assumption that its behavior is probabilistic as well. This formulates the problem in a different way with respect to the adversarial model explained in previous sections.

The most common approach is to model the problem as a Markov Decision Process (MDP) if the evader's position is fully observable, as it is firstly defined by Eaton and Zadeh [25]. If the target's position is not with certainty, Partially

Observable Markov Decision Process (POMDP) can be used, although with their general formulation, the problem can be intractable. In POMDPs the problem is to find a probability distribution over the states, called *belief space*, to actions. The more is the number of states, the higher is the dimensionality of such probability distribution.

The general approach is to employ a discretization of the environment by means of a grid, Roy, Gordon, and Thrun [70] develop a method for compressing the belief space in order to obtain a tractable problem with a single pursuer in a geometric setting, but unfortunately it doesn't scale well with environment and team size. A similar work is done in [62] that employ a sampling of the belief space that provides a compact representation of the state space resulting in a significant speedup with one of the fastest algorithm for point-based POMDPs which rely on an approximation technique for computing a value function that induces the optimal policy. Hollinger et al. [31] manage to solve the scaling issues regarding the size of the team of pursuers by using a receding horizon approximation: their algorithm requires linear time computation for POMDPs that grows exponentially with the horizon length.

Lau, Huang, and Dissanayake [52] defines optimal strategies using a branch and bound algorithm utilizing a novel bounding criteria. An approach using game theory has been applied in environments that could possibly contain obstacles, with players having bounded speed: with two pursuers and one evader, the authors determine that the game is non-zero-sum, and there exists a Nash solution to the one-step non-zero-sum game if the evader has access to the pursuers' information. It is also possible to derive an equivalent zero-sum matrix and compute the Nash equilibrium policies.

At last, Lau, Huang, and Dissanayake [51] study the probabilistic setting on graphs, handling a single pursuer and multiple evaders. Modeling their positions employing a belief distribution, they manage to build a dynamic programming algorithm that reduces the expected time to detect such team of evaders knowing the environment.

The different assumption about the behavior of the target with respect to an evader, does not allow a meaningful comparison between this probabilistic model and the adversarial one we adopt. In particular, in this thesis, the behavior of the target is modeled using a rash evader strategy.

2.3 Differential games

Another approach for formulating pursuit-evasion games is to define a set of *control variables* that can be directly modified by the players such as the rotation or the angles of the wheels of the robots. These variables in turn affect *state variables* that describe the game being played. Typical examples are the position of the players, their speed or orientation with respect to a reference frame (e.g., the Cartesian coordinate system or polar coordinate system). The laws of motion are described using a set of kinetic equations that require the use of derivatives and integrals, hence the name *differential games*.

This formulation has been studied for the first time by Isaacs [35] (and then

extended in [58]), showing that it is a general method that can be applied to different problems. Nevertheless, Isaacs defines a particular pursuit evasion game played in the Euclidean plane without obstacles, the *homicidal chauffeur*, in which the pursuer that sits on a fast but low maneuverable vehicle is against a slower but higher maneuverable running evader, and it has to reduce its distance to a given amount called capture condition. Isaacs develops a method to find a solution to such problems called Isaacs' methodology which is an extension of Hamilton-Jacobi-Bellman equation [9] which provides sufficient conditions for finding saddle-point equilibrium strategies.

Solving such equation in order to obtain a closed form solution is not possible in cases. A different approach is developed using an extension of the so called Pontryagin's Maximum Principle [67] based on the Optimal Control theory. Recently, this approach is used to analyze differential games in a visibility-based context for target tracking [83]. An important result is reported in [44], that seems to suggest that the number of pursuers needed to satisfy the capture condition in a differential game is higher than in a classical visibility-based problem.

The paradigm utilized in differential games is not compatible, because the behavior of the players and the environment are modeled in a different way with respect to the work presented in this thesis and therefore the criteria is not applicable. Moreover, a direct comparison of our approach with the works cited in this section is not possible since the assumptions are different. For instance in the classical adversarial model either geometry or operational research is employed to devise approaches to solve problems, whereas in differential games optimal control theory is used which relies heavily on mathematical analysis.

2.4 Minimax algorithm

The adversarial context of the game results in conflicting goals for the two players that try to maximize their performance given that they are rational agents. If one seeks for optimal moves, a suitable choice is to use the Minimax algorithm. In this sense, we can solve the conflict situation by restating the maximizing problem for one of the two players by *mini*mizing the loss for a worst-case (*max*imum loss) scenario.

The original Minimax algorithm appears in Zermelo [82] but the paper contains some mistakes and imprecisions, and it is corrected and extended also to games which involve more than two players, using an exhaustive formulation, by Von Neumann and Morgenstern [81]. In this thesis, I will consider the simplest case in which there are two players, called MIN and MAX, but I will formalize the problem more precisely in Chapter 3.

One of the beauties of this algorithm (but also other AI algorithms that can be applied to games) is that it can be used for a broad class of games that, at first sight, contains instances that seem unrelated: as long as there is a deterministic, turn-based, two-player, zero-sum game of perfect information (such as chess), the algorithm is suitable. In particular, Russell and Norvig [71] provide a more formal definition:

Initial state. Called S_0, specifies how the game is set up at the beginning. For example, in the chess game it is all the pieces aligned in two rows on the opposite sides of the board or in tic-tac-toe is the 3x3 empty grid.

Turn. Function PLAYER(s) which returns the player that is allowed to make a move given state s, in chess either black or white, in tic-tac-toe either circles of crosses.

Actions. Function ACTIONS(s) which provides a set of legal actions (moves) given the current state s, for instance in chess the pawns can move only forward provided that the path is not blocked by other pieces, or a bishop can move only in oblique and so on. In tic-tac-toe you can put the assigned symbol into an empty cell.

Transition model. Function RESULT(s, a), which defines the new state after the application of action a to state s.

Terminal state. Function TERMINAL-TEST(s) which returns true when in state s the game is over (e.g., a checkmate) or false otherwise. The states that satisfy the test are called terminal states.

Utility function. For each terminal state s the function UTILITY(s, p), called also objective function or payoff function, returns the value of the state s for player p, for instance in chess, being a zero-sum game, if black wins UTILITY(s, BLACK) = 1 while UTILITY(s, WHITE) = -1 or 0 if the state yields a tie situation.

By defining an initial state, the ACTIONS and RESULT functions yield a structure called *game tree* in which nodes encapsulate the states and edges the moves. For instance if at the root node there is the initial state and it's MAX turn, the set of successor nodes consist of one state for each possible move that it can play, to which MIN has to respond. The tree is further built in this alternate manner until the terminal states are reached. At this point, the UTILITY function is applied returning a value from the perspective of MAX since it is the first moving player, which is interested in the states that yield the maximum utility.

Unfortunately, it is often impractical to generate all possible states, and therefore simulating all the game starting from a particular situation is not feasible (for instance in chess there are 10^{40} possible states). Still, we talk about *search tree* which is a particular subset of the original game tree that contains enough nodes that allow the player to choose the next move.

The resulting optimal strategy is composed of a sequence of optimal moves that reach a goal node which contains a terminal state. This sequence of actions can be extracted directly from the game tree using the MINIMAX(n) function that assigns a value for each intermediate node so to allow players to select their best choice according to their policy (either minimize or maximize). The main assumption is that both players are able to play optimally which is not the case in general when

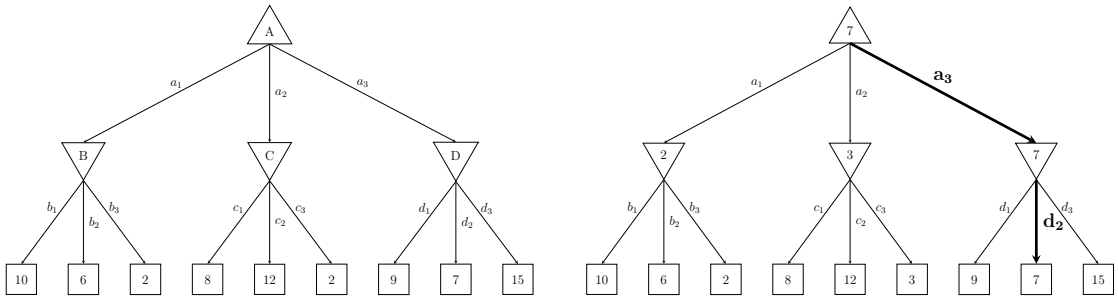


Figure 2.3: An example of minimax game tree, △ indicates the nodes in which player MAX has turn, whereas ▽ indicates nodes in which MIN has to move. On the left hand side there's tree where all nodes expanded till terminal ones □, containing the values of the UTILITY function. On the right hand side the propagation phase has been completed and the resulting optimal actions are highlighted: on this game tree, MAX can get no more than 7.

a human plays against an AI algorithm, but fortunately, this does not apply in this work. The function is defined as following:

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-test}(\text{s}) \\ \max_{a \in \textsc{Actions}(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(\text{s}) = \textsc{Max} \\ \min_{a \in \textsc{Actions}(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(\text{s}) = \textsc{Min} \end{cases}$$

Given the mathematical formulation, we can derive a recursive algorithm called Minimax algorithm, that expands the game tree up to the goal nodes and then in a backtrack fashion propagates up to the root the values that the UTILITY function returns. After all the values propagate, the choice for making the optimal move at the root becomes trivial as the action that leads to the node whose value is the same as the current one at the root, as depicted in Figure 2.3.

Overall, the time complexity of the algorithm is $O(b^m)$ where b is the *branching factor* or the maximum number of new states generated from a state, and m is the maximum depth of the tree. The space complexity can be $O(bm)$ if the successor nodes of each node are generated all at once or $O(m)$ if they are generated one at a time. Given these upper bounds, in real games they can lead to execution times (and memory usage) that are impractical for today machines, therefore some *pruning mechanisms* have to be considered. Such techniques are based on criteria that can safely discard some branches of the game tree that are sub-optimal and so they cannot improve the best solution found so far during the computation. One of the widely used approaches is to use alpha-beta pruning, Section 4.3.2 illustrates an in-depth explanation of how α-β pruning works.

If one is willing to sacrifice optimality, a possible approach is stop the expansion at a given depth d and apply an evaluation function that substitutes the UTILITY function and returns a value that indicates the "goodness" of that state from the perspective of the player that has turn. For instance in chess it might consider the pieces that the player has on the board, their positions, if they are able to attack an opponent piece considering also its importance or if a strong piece (like the queen) is under attack by an opponent one and so on. These values are then propagated

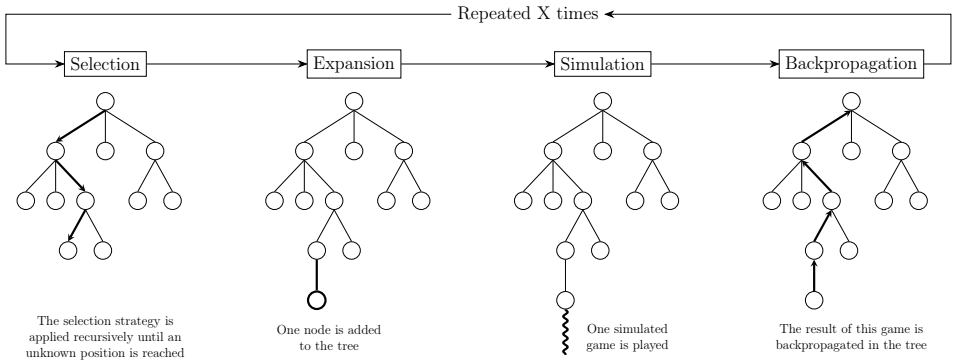


Figure 2.4: Outline of Monte-Carlo Tree Search as showed in [20].

like in the ideal case. If the designer is able to find out an evaluation function that performs reasonably well in quantifying the "goodness" of a state with respect to the player that has turn, the resulting strategy could be not so different from the optimal one.

In the literature there are not so much previous works in the field of pursuit evasion that utilize Minimax, this is another major reason in motivating the study in this thesis. A couple of examples are [69] already presented. Lisý, Bošanský, and Pěchouček [56] make a comparison of Minimax implemented using an iterative deepening framework and Monte Carlo Tree Search in visibility-based pursuit evasion setting. A single evader tries to escape from multiple searching pursuers. The experiments show that both algorithms are valuable in generating a good strategy for the pursuer, while for the evader Minimax is more suitable.

2.5 Monte Carlo Tree Search

The basic version of Minimax raises issues that can be solved in a different way combining random sampling, employing the Monte-Carlo method, with the reliable and precise technique of tree search. The resulting algorithm is called Monte-Carlo Tree Search (MCTS), as it is proposed for the first time by Coulom [22]. The main idea is to select the most promising strategy based on the outcome of a high number of *playouts*, consisting in simulating the game till the end by selecting moves at random.

A sketch of the algorithm appears in Figure 2.4. It can be broken down in 4 stages that can be repeated in sequence as many times as necessary:

(i.) **selection**, starting from the root R, the selection criteria that chooses optimal nodes along the path is applied recursively until a leaf node L that does not satisfies the terminal condition is reached,

(ii.) **expansion**, once L is determined, one or more child nodes are generated and one of them, C is selected,

(iii.) **simulation**, run a simulated game from C until a goal node is encountered,

(iv.) **backpropagation**, the simulated result of the playout is propagated backwards from the leaf node to the root.

In every node there are two values that are updated during execution: one is the ratio between the number of wins and the number of playouts simulated from there, the other indicates how many times the node is visited. The main benefit with respect of the basic Minimax is the ability to adapt well to complex problems exposing high branching factor like Chess or Go [20] because MCTS focuses on the relevant parts of the state space yielding an asymmetric growth of the tree. Further developments in the context of the game of Go lead to the creation of AlphaGo [5] which pairs MCTS and deep learning and is capable of beating for the first time a professional human Go player in a standard 19x19 board and no handicaps.

Another advantage is its contextual freedom, so there is the possibility to disregard an evaluation function which is often difficult to design and most of the times requires a deep level of domain expertise: in the basic MCTS algorithm, only the rules of the games, that are necessary to generate valid moves and the conditions to determine whether a node is terminal or not are enough. Other benefits are the relative simplicity of implementation and the fact that, during execution, it can be interrupted at any time yielding the most promising move found so far.

On the other hand, if the algorithm is facing an expert player, it may overlook a single branch, which may eventually lead to a loss because, being the algorithm random, such branch is not easy to see and might be pruned from the search tree despite being relevant.

Like for the Minimax algorithm, the amount of works in the literature on pursuit-evasion that consider these algorithms is scarce, the only relevant paper is again [56], already cited in the previous section. This thesis tries to fill the gap in this sense by doing a more in depth comparison in different environments and a single pursuer and evader.

2.6 Computational geometry

Algorithms that executes on a geometrical environment often rely on a particular branch of Computer Science called Computational Geometry. It is also used in Computer-Aided Design (CAD), Integrated Circuit design, 3D Computer Vision, and so on.

A typical computational geometry library provides a set of algorithms and data structures to solve a wide variety of problems, focusing on efficiency and speed. Common classes of problems addressed by these libraries are: convex hulls, linear programming, Boolean and set operations on polygons, as well the ability to make queries on geometrical structures like point location, ray tracing, and so on.

Regarding the construction of visibility polygons which is a crucial aspect in the related class of problems cited earlier, some algorithms have also been studied in the academia. An efficient method is implemented by Asano [8] specifically for polygons with holes. Concerning simple polygons, the first work about the topic is [55], later corrected and extended in [40]. Recently, a more general algorithm that results in a dramatic speedup uses a preprocessing phase based on a constrained triangulation of the input polygon [17].

The usage of CGAL for the implementation of the algorithms for pursuit evasion in geometrical context seems not widespread, the only notable works are [26, 69].

This stimulates research about the topic, in particular this thesis tries to determine whether it is a suitable choice in terms of efficiency and general applicability to such problems.

2.7 Wrap up

Table 2.1 (in the next page) summarizes the works appearing in this chapter. The selection of the columns reflects the criteria that Section 2.1 introduces. Differential games are not listed because they are modeled using a completely paradigm that doesn't fit in the classification.

Table 2.1: List of pursuit-evasion games.

Paper	Setting	Environment	Information available to players	Sensing capabilities	Movement capabilities	Number of pursuers	Number of evaders	Capture condition
[57, 72]	Adversarial	Geometrical	Global visibility	N/A	Same speed	1	1	Same position
[11]	Adversarial	Polygonal with holes	Global visibility	N/A	Same speed	k	1	Same position
[50]	Adversarial	High dimensional	Global visibility	N/A	Same speed	$d+1$	1	Same position
[45]	Adversarial	Three dimensional	Global visibility	N/A	Same speed	3	1	Same position
[10]	Adversarial	Two dimensional geometric topological spaces	Global visibility	N/A	Same speed	> 1	1	Same position
[3]	Adversarial	Convex Euclidean Domains	Global visibility	N/A	Same speed	$d+1$	1	Same position
[4]	Adversarial	Geometrical satisfying CAT(0)	Global visibility	N/A	Same unitary speed	1	1	Within a certain range
[12]	Adversarial	Polygonal with holes	Global visibility	N/A	Same unitary speed	> 1	1	At least one in the same position
[6]	Adversarial	Polygonal with holes	Global visibility	N/A	Same speed	> 1	1	At least one in the same position
[77]	Adversarial	Simply connected polygonal	Local visibility	Pursuer has k flashlights	Evader can move arbitrarily fast	1	1	Only detetion
[74]	Adversarial	Polygonal with holes	Local visibility	Omni-directional with unlimited range	Evader can move arbitrarily fast	> 1	1	Same position
[79]	Adversarial	Simply connected polygonal	Local visibility	Omni-directional with unlimited range	Speed bounded but may differ between players	1	1	Same position
[75, 76]	Adversarial	Simply connected polygonal	Local visibility	Omni-directional with unlimited range	Speed bounded for pursuer, unbounded for evader	1	1	Same position
[69]	Adversarial	Simply connected polygonal	Local visibility	Omni-directional with unlimited range	Same bounded speed	1	1	Same position
[15]	Adversarial	Geometrical	Local visibility	Omni-directional with limited range	Same bounded speed	1	1	Within a certain range
[42]	Adversarial	Geometrical	Local visibility	Pursuer can detect the evader, but their exact distance is unknown	Same bounded speed	1	1	Same position
[26]	Adversarial	Polygonal	Local visibility	Sensors have unlimited range but limited FOV and angular aperture	Speed bounded for pursuer, unbounded for evader	1 (extension to > 1)	1	Only detection
[28, 53]	Adversarial	Polygonal	Local visibility	Omni-directional with unlimited range	Speed bounded for pursuer, unbounded for evader	$\geqslant 1$	1	Only decetion
[37] (Locating game)	Adversarial	Simply connected polygon	Local visibility	Omni-directional with unlimited range	Speed bounded for pursuer, unbounded for evader	1	1	Only detection
[37] (Capture game)	Adversarial	Simply connected polygon	Local visibility	Omni-directional with unlimited range	Same bounded speed	1	1	Same position
[13, 59]	Adversarial	Polygonal with holes	Local visibility	Omni-directional with limited range	Same bounded speed	$\geqslant 1$	1	Only detection
[64]	Adversarial	Polygonal	Local visibility	Omni-directional with limited range	Speed bounded for pursuer, unbounded for evader	1	1	Only detection
[54]	Adversarial	Three dimensional	Local visibility	Omni-directional with unlimited range	Unbounded	1	1	Only detection
[43]	Adversarial	Simply connected polygon	Local visibility, environment unknown	Omni-directional with unlimited range, additional sensors depending on the model adopted	Evaders can move arbitrarily fast	$\geqslant 1$	1	Only detection
[24]	Adversarial	Geometrical	Local visibility	Hardware with limited range and FOV	Evaders can move arbitrarily fast	> 1	> 1	Only detection
[60]	Adversarial	Polygonal X-monotone	Only the evader has global visibility	Omni-directional with unlimited range	Same bounded speed	1	1	Within a certain range
[65]	Adversarial	Connected graph	Only the evader has global visibility	Pursuer can perceive only in its vertex	Evader is arbitrarily fast, can teleport	> 1	1	Same vertex
[61]	Adversarial	Graph-based	Global visibility	N/A	Unbouded	1	1	Same vertex

Paper	Setting	Environment	Information available to players	Sensing capabilities	Movement capabilities	Number of pursuers	Number of evaders	Capture condition
[27]	Adversarial	Graph-based	Global visibility	N/A	Unbouded	k	1	At least one cop in the same vertex
[2]	Adversarial	Graph-based	Global visibility	N/A	Unbounded	1 (extension to > 1)	1	Same vertex
[1]	Adversarial	Graph-based	Local visibility	Players can perceive only in their vertex	Unbounded, jumps allowed	1	1	Same vertex
[38]	Adversarial	Graph-based	Local visibility	Players can see the nodes adjacent to their position	Unbounded	> 1	1	Same vertex
[39]	Adversarial	Graph-based	Evader has global visibility, pursuer has local	Pursuer can see within distance at most k	Unbounded	1	1	Same vertex
[47, 48]	Adversarial	Graph-based, tree-based	Evaders are omniscent, wheras pursuer aren't	Pursuer have sensors with limited range	Evaders have unbouded speed	n	m	Same vertex
[30]	Adversarial	Graph-based	Environment is known to both players	Pursuer can perceive in its vertex	Evaders have unbouded speed	k	1	At least one pursuer in the same vertex
[16]	Adversarial	Graph-based	Evader knows in advance the strategy of the pursuer	Pursuer can perceive in its vertex	Evader moves at specific time, pursuer asks admittance to a vertex	1	1	Same vertex
[49]	Adversarial	Graph-based	Global visibility	N/A	Players can only move to adjacent vertices	> 1	1	Same vertex
[7]	Adversarial	8-connected grid map, modeled as a graph	Local visibility	Pursuer is able to detect the evader when it is in the same cell	Players can only move to adjacent vertices	1	1	Same cell
[32]	Non-Adversarial	Graph-based	Evader knows the stregy of the pursuers	Pursuers can perceive only in their cells using perfect sensors	Only evader is arbitrarily fast	k	1	Same vertex
[70]	Probabilistic	Geometrical	Position of pursuer is known, a belief distrubion models the position of the evader	Evader becomes visibile if the pursuer is within certain distance	Evader has a random strategy with known fixed speed	1	1	Same position
[62]	Probabilistic	Polygonal with holes	Position of pursuer is known, a belief distrubion models the position of the evader	Evader is invisible unless the position is the same	Same bounded speed	1 (extended to 2)	1	Same position
[31]	Probabilistic	Geometrical divided into cells that induces a graph	The movement model of the evader as well its position is known	N/A	Same bounded speed	1 (extended to k)	1	Same vertex
[52]	Probabilistic	Polygonal divided in cells	Location of the evader modeled using a probability distribution	N/A	Same bounded speed	1	1	Same cell
[51]	Probabilistic	Graph-based	A belief distrubion models the position of the evader	N/A	Same bounded speed	1	m	Same vertex

Chapter 3

Pursuit-evasion in simple polygons

This chapter provides a more in-depth explanation of the pursuit-evasion problem that this work examines. Section 3.1 gives a formal definition of the problem studied in this thesis, in Section 3.2 I state the hypotheses and assumptions, while Section 3.2.1 illustrates the main contributions of this work. Finally, Section 3.3 focuses on a description the characteristics that contextualize the problem analyzed in this thesis.

A lot of previous research works in the visibility-based pursuit-evasion focus into finding a complete algorithm that guarantees capture or detection in the particular case studied, disregarding any form of optimality in the motion plan computed by the method. As an important remark, in this work, the concept of optimality applies not only to the pursuer, but also to the evader: this makes the problem somewhat difficult. Nevertheless, there are some works that concentrates efforts in this direction, as described in Section 2.1.1. In particular, [69], which is the paper that this thesis tries to extend, and [75, 76] that focus only on detection rather than capturing.

The main objective in this work is to design and implement an algorithm that is able to perform both detection and capture in an optimal way with respect to precise criteria. The algorithm is able to clear the environment, allowing the pursuer to plan a motion strategy in order to first detect an escaping evader and then elaborate the sequence of moves in order to ensure capture. During escaping, it might happen that the evader is able to hide again. In this particular case, the algorithm needs to handle the situation by falling back to the detection phase. Whenever the pursuer detects the evader again, the capture phase resumes.

3.1 Formal definition

As already stated in Chapter 2 the pursuit-evasion problem can be studied in many different situations and levels of abstractions. It is virtually impossible to design a method that is able to perform well in any possible case, therefore a number of assumptions has to be made. The set of hypotheses and assumptions must be defined carefully in order not to simplify too much the problem. On the other hand, the set has to ensure that the problem is defined as clearer and specific as possible in order to be meaningful.

This thesis studies the pursuit-evasion problem by modeling the environment as a simply-connected polygon $\mathscr{P}$ without obstacles. Both pursuer and evader know the environment. A simple polygon is a closed chain of line segments that do not intersect each other. Given this definition, a simple polygons adhere to these properties: *(i)* the interior, i.e., the region enclosed by a polygon always has a measurable area, *(ii)* the segments (called edges) that constitute the closed chain, meet only at their endpoints, called vertices, *(iii)* exactly two edges meet at each vertex, *(iv)* the number of edges always equals the number of vertices.

Without loss of generality, the robots can be assumed to be points in the plane that move at the same speed which can be considered unitary.

The game is played in turns in an alternate manner allowing a discretization of the time variable t. It is possible to define a *time step* in which one of the two player has turn. The pursuer makes the first move (which is considered at time step 0) and then at every successive even time step, whereas the evader has turn at every odd time step.

Given the time step t, it is possible to define the current position of both players, in particular $p(t)$ indicates the location of the pursuer at time step t. Similarly, $e(t)$ denotes where the evader is located inside the environment.

Each robot is equipped with an omni-directional sensor with unlimited range that is capable of perceiving the surroundings at any given time. The players use their respective sensors primarily to detect the presence of the adversary. This enables the robots to have a line-of-sight visibility which can be formally defined as the set of points that can be joined from a point $q \in \mathscr{P}$ with a line that doesn't cross the boundary of $\mathscr{P}$ itself. The resulting set is called *visibility region* and denoted as $\mathrm{VR}(q)$.

Using this notation it is possible to say that both robots can see each other if and only if $p(t) \in \mathrm{VR}(e(t))$, likewise $e(t) \in \mathrm{VR}(p(t))$. Every time a robot moves inside the environment, it makes a perception using its sensor, resulting in a new visibility region. The prior knowledge can be still used provided that the visibility regions cleared in previous time steps are not recontaminated again. The *cleared region* encapsulates this concept and it is defined as $\mathrm{CR}(p(t)) = \bigcup_{s=0}^{t} \mathrm{VR}(p(s))$, an analogous definition applies for the evader with proper terms substitution. Note that, at the beginning, when $t = 0$, $\mathrm{CR}(p(0)) = \mathrm{VR}(p(0))$ and $\mathrm{CR}(e(0)) = \mathrm{VR}(e(0))$. From the point of view of a player, a region is contaminated when it is suspected that its adversary is located in that region.

During the detection phase, each player does not know the position of its adversary. In particular, the pursuer can reason on its current knowledge about the position of the evader, which is a set of points denoted as $E(t) = \mathscr{P} \setminus \mathrm{CR}(p(t))$; likewise, from the evader's perspective, its knowledge about the position of the pursuer is $P(t) = \mathscr{P} \setminus \mathrm{CR}(e(t))$.

The winning condition for the pursuer is satisfied whenever at a finite time step t^*, that the pursuer aims at minimizing, its location is the same as the evader, which tries to maximize t^*, that is, $p(t^*) = e(t^*)$. Conversely, the evader wins if it is able to escape indefinitely.

The evader plays the so called *rash evader* strategy which consists of standing still in its position until the pursuer becomes visible. If line-of-sight with the pursuer

is established, the evader seeks to move towards another point of the map. Note that from the point of view of the evader, the choice of sitting in a vertex dominates the selection of other points because otherwise in the latter case the evader could be spotted more easily. This strategy models some practical scenarios. For instance in a search and rescue context during a fire, a person in a dangerous situation tends not to move. As soon as he/she notices a rescuer, the person may mistakenly think that is a threat and tries to escape. Nevertheless, for an in-depth explanation of this concept see Section 3.3.2.

3.2 Research hypotheses

The key motivation of this work is to focus on optimality which most of the existing studies don't consider. This leads to the following research questions:

$\mathcal{H}_1$ Is the proposed algorithm optimal? Meaning that is able to determine a motion plan that minimizes a particular objective function that this setting considers relevant.

$\mathcal{H}_2$ Can both subproblems be solved optimally? The algorithm solves two distinct subproblems: one is detection and another is capture. The detection phase is optimal if prevents the recontamination of an already cleared region minimizing the possibility that the pursuer visits more than one time a region. The capture phase is optimal if the number of time steps is minimized respect to the pursuer or maximized for the evader. In order to achieve general optimality and proving $\mathcal{H}_1$, both of these subproblems shall be solved optimally.

$\mathcal{H}_3$ Is the complexity, both in time and space, of the algorithm reasonable? If it is not able to determine an optimal strategy within a certain amount of time, the algorithm is aborted and the most promising action is selected. The same policy is applied whenever the amount of memory consumed exceeds a certain threshold. This ensures that the stability of the system is not compromised.

$\mathcal{H}_4$ Is the algorithm robust with respect to the environments? It should work on different kinds of realistic maps such as offices composed of many rooms or with long corridors or even a combination of both characteristics.

Coupled with the formal definition described in Section 3.1 the aim of this thesis is to design and develop the algorithm and then assessing its performance. The evaluation is performed using an ad hoc simulation program. This simulations shall verify the hypotheses $\mathcal{H}_1 - \mathcal{H}_4$. Different combinations of the input parameters have to be taken into account in order to stress the algorithm in a variety of different cases and verify its robustness.

3.2.1 Main contributions of this thesis

Given the problem defined above, the main contributions of this thesis are: *(i)* an optimal algorithm for a single pursuer that is able to detect and capture an escaping evader in a simply connected polygonal environment without internal

obstacles, *(ii)* a novel algorithm for the detection phase that uses the visibility-based polygonal decomposition introduced in [69], *(iii)* additional pruning mechanisms that speedup the computation for the capturing phase using Minimax with respect to the original techniques introduced in [69] as well, *(iv)* simulations on different maps and parameters to test the robustness of the whole algorithm, *(v)* a comparison between the optimal strategy computed by Minimax algorithm against the solution determined by a Monte Carlo Tree Search algorithm.

3.3 The case study

In this section, I will briefly describe the peculiar characteristics that contextualize more precisely the problem that this thesis tries to solve, in particular why it is reasonable to use simple polygons for modeling indoor environments and an explanation that motivates the choice of the rash evader strategy.

3.3.1 Indoor maps as simple polygons

As I have already mentioned in Section 3.1, the simulated searches use simply-connected polygonal indoor environments. As Chapter 2 shows, a lot of previous works model the environment in the same way, therefore the choice of using this type of polygons is reasonable. Although internal obstacles are excluded, they can model a variety of maps, for instance: an office consisting of many rooms and a single corridor, a university department with longer corridors, or even a combination of both. Figure 3.1 **(a)** shows an example of such maps.

Another advantage is that, in computational geometry, there are some common tasks that receive as input simple polygons. Chapter 4 shows that some of them are also utilized in the proposed algorithm.

- Point in polygon, which, given a simple polygon $\mathscr{P}$ and a query point q, the algorithms tells whether q lies inside the interior of $\mathscr{P}$ or not.
- Formulas to compute the area, which are simple and efficient.
- Polygon union, a crucial aspect for computing the cleared region.
- Polygon intersection, another fundamental concept that helps the representation of contaminated regions.

3.3.2 Rash evader setting

The decision of modeling the evader's behavior using a rash evader strategy is not unreasonable with respect to a *general evader* one. The latter allows the evader to move freely at every turn. In most practical environment one can safely assume that the two strategies have more or less the same power. For instance in Figure 3.1 **(b)**, the pursuer is moving along a corridor in an effort to capture the evader. Since line-of-sight is established, the evaders makes a move and hides in either one of the two pockets v or v' losing the pursuer. Now both players don't have knowledge about each other, therefore the pursuer falls back in the detection phase. At this

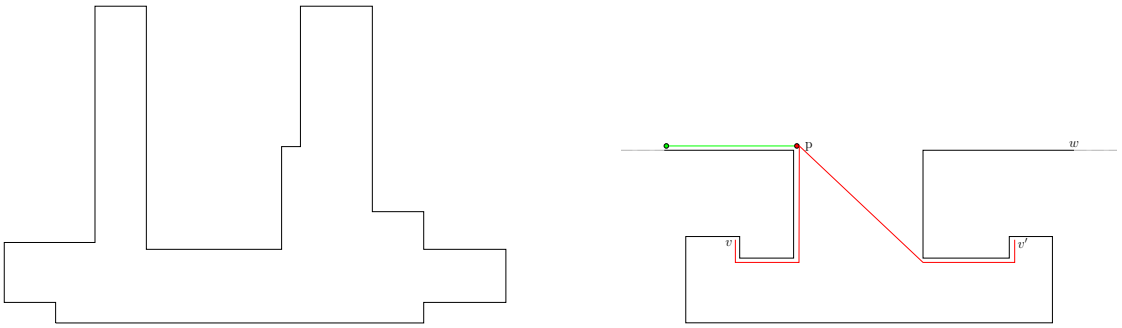


Figure 3.1: **(a)** An example of a simply-connected polygon map. **(b)** Even assuming the worst case position of the evader (red point), it doesn't gain any advantage using a general strategy.

point, the evader may try to move and reach w, exiting the room and getting to the corridor once again, but with the risk of being captured by the pursuer. Suppose that the evader chooses v and the pursuer decides to clear position v'. In the mean time, the evader can move to w and successfully escape, provided that is able to time correctly its movements. Unfortunately, this is not easy since, for instance, the pursuer can wait t time steps before starting to clear, managing to successfully intercept its adversary.

Additionally, the choice of employing the rash evader strategy reduces the complexity of the first phase of the algorithm since once a region of the map is cleared, it is ensured that the portion of the map can be no longer recontaminated unless the evader is detected, triggering the second phase. Moreover, since the map is closed there's the guarantee that eventually the pursuer can detect the evader.

Chapter 4

Solving the problem

This chapter explains in detail the algorithm that solves the problem that Chapter 3 formalizes. Section 4.1 shows an overview of the method, along with an explanation of the partitioning techniques used. Section 4.2 presents a complete description of the first phase of detection. Section 4.3 and Section 4.4 delve into the capturing part. The former using the Minimax algorithm, while the latter employs Monte-Carlo Tree Search. The two different approaches are compared in Chapter 6.

4.1 Algorithm overview

This section gives an overview of the proposed algorithm that is described in detail in the next sections. Moreover, a detailed explanation of the methods utilized for partitioning the environment is given. These techniques are fundamental because they are closely related to the actual algorithm. They are primarily used for the state-space representation but during the computation of the motion strategy they are used to make some estimations that help the players reasoning about what to do.

4.1.1 Splitting the algorithm in two phases

As introduced in Chapter 3, the computation of the motion strategy of the pursuer can be decomposed in two distinct phases. The first one deals with clearing the environment in order to find out the position of the evader. Then, as soon as it is discovered, the second phase starts and takes into account the adversarial model of the situation in order to be able to devise the sequence of moves that eventually leads to the capture of the evader.

In the first phase, the pursuer does not know the position of its adversary, likewise for the evader. The evader stays put behind a vertex during this phase, according to the rash evader strategy explained in Section 3.1 and Section 3.3.2, whereas the pursuer tries to extend its cleared region by planning the actions carefully.

The crucial aspect for the pursuer is to always be aware about the portions of the polygon that still need to be explored. This can be computed in an easy way as I will show in Section 4.2. The motion plan should take into account the

distance between the pursuer's current position and the regions that need to be cleared. This is achieved by combining a particular method for partitioning the environment explained in Section 4.1.2, together with an algorithm that computes the shortest path between two points using a grid decomposition.

Regarding the second phase, the model imposes reasoning about the sequence of actions that both the pursuer and the evader need to perform. These two sequences strongly depend on each other. That's why an adversarial search algorithm is employed. In particular, since guaranteeing optimality is the main objective, Minimax is a reasonable choice, although it requires some degree of domain knowledge to extract its full potential. This approach is described in detail in Section 4.3.

Section 2.4 shows that the time complexity of Minimax is exponential in the branching factor and depth of the search tree. In realistic scenarios, even by applying speed ups, the algorithm may require too much execution time, or may not terminate at all. An interesting idea is trying to execute the Monte Carlo Tree Search (MCTS) algorithm. The goal is to understand whether it is reasonable to sacrifice some optimality while gaining some speedup in the execution time. Section 4.4 explains how MCTS is adapted to the pursuit-evasion problem addressed in this thesis, whereas the comparison is highlighted in Chapter 6 along with other experiments.

4.1.2 Partitioning the environment

In general, in robotics, an environment is partitioned in cells allowing plan computation to be more efficient. Such a technique can be used also in pursuit-evasion problems.

A typical approach is to tile $\mathscr{P}$ into cells G^e such that each point $q \in \mathscr{P}$ belongs to a single cell of the grid. The cells have equal size with edge e and can be either free or occupied. Starting from the grid decomposition, a corresponding matrix is derived which encodes for each cell the possibility of either being reached or not. Since the matrix has the same dimensions of the bounding box that surrounds the environment, it might happen that some cells fall off the boundary of the map but are still present in the matrix, therefore they have to be marked as not reachable.

Moreover, if the length of e is not a multiple of the coordinates of a particular boundary, small regions near the contour of the map are not covered and must be properly marked as well.

The position of the players is assumed to be in the center of a cell, with the next possible actions as the movements towards the center of cells within distance $k = \frac{\nu}{e}$. If $\nu = e$, then the adjacent cells are considered, otherwise if $\nu > e$, furthest cells are taken into account.

This decomposition is mainly used during the capturing phase: in this situation it is mandatory to take into account the real speed of the two players in order to reason about a physically feasible strategy.

A grid-based decomposition is used also by algorithms to compute the shortest path between two locations in the environment. The aim is to fill, in a Breadth First fashion, a matrix with the same dimensions as the first one introduced. This second matrix contains the distances from a starting point. After the computation is done, the value of the matrix cell that contains the target point is returned. Figure 4.1 shows an example of a matrix projected onto a portion of a sample map.

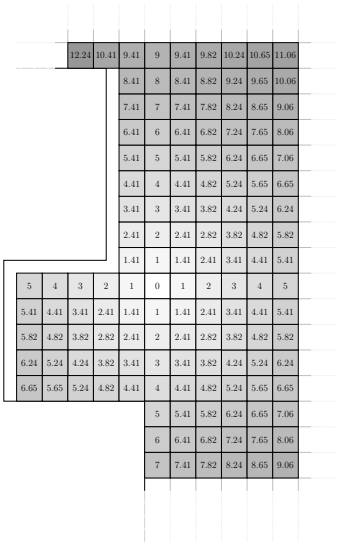


Figure 4.1: Example of a partial computation of the distance matrix using the grid decomposition.

Notice also the fact that the cost of moving in oblique is slightly higher than going horizontally or vertically. This is reasonable, since by moving in oblique the robot has to cover more distance when going from one cell to another. The output of this algorithm can be used as heuristic function, namely an estimate on how many actions are needed by one player to reach a candidate location.

The main issue with partitioning the environment with a grid is the following: if the length of e is too small, a high number of states are generated that can slow down excessively the computation of a solution. Another crucial aspect is that it doesn't encode any visibility information.

To overcome this problem, a different approach is used which takes into account the geometry of the environment. This allows to dramatically reduce the number of cells and therefore states and actions. The idea is to consider line segments that completely lie inside the polygon and join each pair of visible vertices, then extend such line, called *inducing line* till the contour of the map. The intersection between lines also with the boundary of the environment results in the creation of cells. This partitioning of the environment is called *visibility-based decomposition* and by construction it generates convex cells. An example of such decomposition is shown in Figure 4.2.

With this technique, the state is encoded with a point that is derived from a discretization of the inducing lines, and the actions are the movements towards the discretized borders of the cells that are adjacent to the one in which a robot belongs to.

This decomposition is used solely for the first phase, since a single movement can cover a distance potentially higher than that allowed by the maximum speed of the robot. Nevertheless, this is not a problem since the pursuer is the only one in charge of moving while its adversary plays the rash evader strategy. Moreover, by construction, the detection happens only if the pursuer reaches a point on one of the inducing segments.

Even if this method ensures a coarser discretization of the environment with

respect to the one provided by a grid, when computing the successive states, it might happen that the adjacent discretization points fall in the same grid cell where the pursuer resides because the inducing lines are too small. Therefore, these points don't reduce the heuristic function and in this case, the pursuer resorts to the grid decomposition.

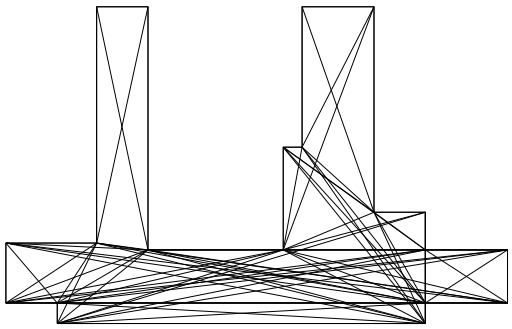


Figure 4.2: Example of visibility-based decomposition of a map.

4.2 Phase 1: searching the evader

The first phase of the algorithm deals with clearing the environment, the procedure can be synthesized as follows (Algorithm 1):

(i) by using the cleared region of the pursuer, the contaminated nodes, where it is suspected that the evader may reside, are constructed (Algorithm 2),

(ii) the nodes are inserted in an empty frontier and sorted in increasing order of distance from the pursuer's current position (Algorithm 3),

(iii) the nearest node is selected, and among the points that are eligible to clear such node, the one that maximizes the most the area that is discovered by "jumping" on it is selected,

(iv) by using a greedy strategy, among the adjacent points of the cells generated by the appropriate decomposition, the one that minimizes the most the distance from the pursuer to the target point is chosen, and a move is perfomed,

(v) repeat the algorithm by going to step *(i)*, or stop if the evader is detected or if the global timeout expires.

The rest of this section describes in the detail the algorithm by providing also the pseudo-code to better illustrate its functioning.

4.2.1 Search as a graph covering problem

Algorithm 1 describes, using pseudo-code, what is the strategy for selecting the points and therefore the sequence of actions to clear the environment. The function expects the initial state and the frontier as input since it could possibly

be called also during phase 2 when the players lose line-of-sight visibility. For now let's consider the beginning of the search, i.e., at time step $t = 0$. The state of the algorithm, called currentState is a quadruple defined as $(p(0), e(0), \mathrm{CR}(p(0)), 0)$ and frontier $= E(0)$ with each region that needs to be cleared encapsulated in a node.

The outermost loop continues as long as there is time for performing the computation, there are still nodes in the frontier and the pursuer has not detected yet the evader (and vice versa). If this Boolean condition is satisfied, the pursuer selects the nearest contaminated region, which is the first element of the frontier managed as a priority queue. The pursuer wishes to clear this node and therefore it has to determine a suitable target point in order to do so.

For each region, it is possible to compute the nearest line segment with respect to the current position of the pursuer. The computation is explained in Algorithm 2. Given the segment, its point discretization is considered. The points belonging to the contour of the map, or that are outside the reachable cells of the grid decomposition are discarded. The criteria for selecting the target point is to maximize the area of the hypothetical cleared region that would be created if the pursuer moved to the discretization point being evaluated. This is computed from line 7 to line 15 of Algorithm 1.

Now the pursuer has to determine the adjacent point with respect to its current position, which allows to minimize the most the distance from a target point. For this reason, the visibility-based decomposition is used again paired with the heuristic function. If the discretization is too fine, it might happen the problem that the adjacent points reside in the same cell of the current position, leading to indistinguishability, as explained in Section 4.1.2. In this particular case, the pursuer computes its adjacent points falling back to the grid, then again evaluates the goodness of each point using the heuristic function, as performed from line 23 to 38.

If no valid points are found, the algorithm stops with an error. Otherwise the pursuer moves to the designated point (nextPoint). After making a perception, the cleared region gets updated, since the shape of the state regions may be changed. For instance a region could be split or cleared, therefore the frontier must be updated as well. After a movement, the time step is increased.

This algorithm guarantees that the pursuer clears all the contaminated regions and eventually detects the evader. Recontamination is not possible because, intuitively, since the evader is playing the rash strategy, it moves if and only if line-of-sight is established, thereby concluding the detection phase. Before exiting from the function, the final position of the pursuer is assigned to the input parameter, which can be considered as a pointer to some variable, also the total cost is returned. This is when the second phase starts and how this is handled will be explained in the next sections.

Algorithm 1 Planning the motion strategy in the detection phase.

```
1: function PHASE1(pursuerPos, currentState, frontier)
2:     while hasTime ∧ frontier is not empty ∧ evader is not detected do
3:         currentNode ← frontier.front()
4:         Q ← currentNode.nearestSegment().points()
```

```
5:        maxArea ← −1
6:        targetPoint ← null
7:        for all q ∈ Q do
8:            if q ∈ P_𝒫∨ Map.cellStatus(q) then
9:                continue
10:           end if
11:           if A(CR(p(t)) ∪ VR(q)) > maxArea then    ▷ A(x) is the area of x
12:               maxArea ← A(CR(p(t)) ∪ VR(q))
13:               targetPoint ← q
14:           end if
15:       end for
16:       adjacentPoints ← Map.adjacentVisibilityPoints(currentState.pursuerPos())
17:       minDistance ← Map.computeDistance(targetPoint, currentState.pursuerPos())
18:       nextPoint ← null
19:       found ← false
20:       gridUsed ← false
21:       if minDistance = 0 then nextPoint ← targetPoint
22:       else
23:           for all adjPoint ∈ adjacentPoints do
24:               if adjPoint ∈ P_𝒫∨ Map.cellStatus(adjPoint) then ▷ P_x denotes
    the set of points of the perimeter of polygon x
25:                   continue
26:               end if
27:               adjDistance ← Map.computeDistance(targetPoint, adjPoint)
28:               if adjDistance < minDistance then
29:                   minDistance ← adjDistance
30:                   nextPoint ← adjPoint
31:                   found ← true
32:               end if
33:           end for
34:           if found = false ∧ gridUsed = false then
35:               adjacentPoints ← Map.adjacentGridPoints(currentState.pursuerPos())
36:               gridUsed ← true
37:               goto 23
38:           end if
39:           if found ≠ true then
40:               error("Cannot find an adjacent point that reduces the distance
    to the target point")
41:               return -1
42:           end if
43:           currentState.moveTo(nextPoint)
44:           frontier.update()
45:       end if
46:   end while
47:   pursuerPos ← currentState.pursuerPos()
48:   return currentState.cost()
49: end function
```

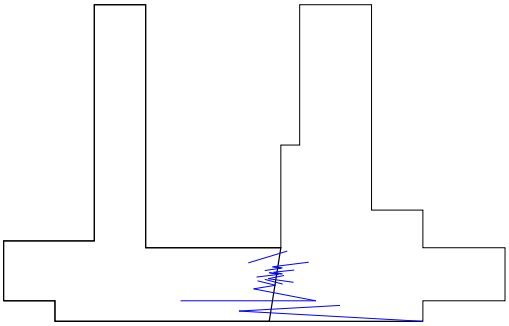


Figure 4.3: Intersecting a contaminated region's edge with the visibility-based decomposition.

4.2.2 Constructing a node

Every time the frontier gets updated, nodes that encapsulate the contaminated regions are constructed. Algorithm 2 delineates the pseudo-code of the constructor for a particular node. The procedure receives in input the current pursuer position and an object that represents the set of edges that constitutes the boundary of the region. The aim of this function is to calculate an heuristic value that is used as the sorting criteria in the frontier, and the nearest segment of the visibility-based decomposition with respect to the pursuer.

The outermost loop iterates over all edges of the region's boundary, discarding the ones that are in common with the contour of the environment. Next, the intersection between an edge and the inducing lines of the visibility-based decomposition is performed, generating the set of line segments that cross the region's edge being considered. Figure 4.3 shows an example result of such computation.

Finally, by iterating over the discretization points for each visibility segment, the heuristic function is called returning the distance of every point from the pursuer's current position. Among all the points, the nearest is considered and its distance is assigned to a field called heuristics.

The visibility segment to which the nearest point belongs to is assigned to another private field called nearestSegment that is also publicly accessible and already utilized as showed in Algorithm 1.

Algorithm 2 Constructing a new graph node

```
1: function GraphNode(pursuerPos, regionPolygon)
2:     heuristics ← +∞
3:     nearestSegment ← null
4:     for all edge ∈ regionPolygon.edges() do
5:         if edge ∈ 𝒫.edges() then ▷ Discard edges that are in common with the environment
6:             continue
7:         end if
8:         visibiliySegments ← edge.intersects(visibilityPartition.segments()) ▷ Consider only the inducing lines that crosses this edge
```

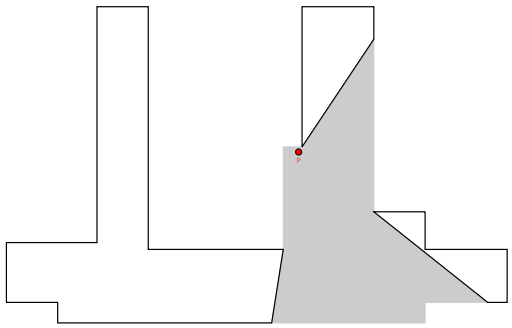


Figure 4.4: Polygonal difference between the environment and cleared region (colored in gray).

```
        for all segment ∈ visibilitySegments do
            for all q ∈ segment.points() do
                distance ← Map.computeDistance(pursuerPos, q)
                if distance < heuristics then
                    nearestSegment ← segment
                    heuristics ← distance
                end if
            end for
        end for
    end for
end function
```

4.2.3 Updating the frontier

Another relevant algorithm is the frontier.update() method that, given the current state, uses the visibility region to construct a set of contaminated nodes and then sorts them. Algorithm 3 shows this method, firstly it computes the set using a polygonal difference between the environment and the pursuer's cleared region. Then, for each node, the GraphNode constructor (showed in Algorithm 2) is called, filling the necessary node's fields in order to sort them at the end of the Update function. The nodes are sorted in increasing order of heuristic value. Figure 4.4 shows an example of what happens when a polygonal difference is computed given the map and the pursuer cleared region, the resulting three contaminated regions are highlighted with a thicker black border.

Algorithm 3 Updating the frontier used in the detection phase

```
1: function Update
2:     list.clear()
3:     contaminatedRegions ← Map.environmentPolygon() \ CR(p(t))    ▷ Set of
   polygons one for each region that is not yet present in the cleared region of the
   pursuer
4:     for all contaminatedRegion ∈ contaminatedRegions do
5:         node ← GraphNode(currentState.pursuerPos(), contaminatedRegion)
6:         list.push(node)
7:     end for
```

```
8:      list.sort()
9: end function
```

4.2.4 Complexity of the algorithm

The seek of determining in which complexity class the decision problems of clearing environments belong is still ongoing. For instance, [47, 48] manage to prove that the decision problem of determining the minimum number of searchers necessary to optimally clear a graph environment is NP-hard.

In the setting studied in this thesis, the algorithm generates the strategy for clearing the environment using a greedy approach that minimizes the most the distance towards the discretization points that belong to segments that are partially visible. Since the evader is playing the rash strategy this ensures that the number of times in which the pursuer passes on a particular point is minimized. Besides this kind of optimality criteria, the possibility to demonstrate that it minimizes the number of time steps in every possible setting, seems somewhat difficult as [47, 48] point out.

The algorithm can be seen as a Depth-First-Search (DFS) one that explores deep branches of a tree using a greedy strategy. The geometrical nature of the problem induces an implicit graph that the proposed algorithm traverses by computing the successive nodes either using the visibility-based decomposition or the grid. It is well known in the literature that a vanilla DFS on an unknown graph has a competitive complexity of 2 against the optimal full knowledge algorithm. Kamphans [41] further improve the efficiency of the algorithms on simple polygons decomposed using a grid, where every strategy using such partitioning technique yielding C cells needs at least $\frac{7}{6}C$ steps to clear the environment. The fact that in this thesis a technique for compressing the state space is employed suggests that the bound could be further tightened. The competitive complexity can be also improved if some degree of knowledge about the graph is given to the pursuer as Dessmark and Pelc [23] show. Regardless of the various speedup techniques concerning competitive complexity, the computational complexity of DFS is $O(b^d)$ where b is the branching factor and d is the depth of the search tree.

4.3 Phase 2: capturing the evader – Minimax

The first method presented for performing the actual capture of the evader is the already mentioned Minimax. It uses the state-space induced by the grid decomposition coupled with the heuristic function based on the same partitioning method, which is used to estimate the distance between the players and to select nodes that are more favorable for the player that has turn. Such choice should allow to reach a goal node more quickly, being Minimax a Depth-First-Search algorithm. Several methods to improve the performance are presented. The potential impact about the optimality of Minimax is thoroughly discussed for each proposed technique.

4.3.1 Setting up the search problem

Following the formal definition of the Minimax algorithm given in Section 2.4, it is possible to describe the capture phase. In particular, given a time step t, the corresponding state $s(t)$ is defined as a triple $((p(t), e(t), h(t)), cp)$ (see Section 3.1 for the notation). Besides the current position of the players already defined, $h(t) = d(p(t), e(t))$ denotes the distance computed using the heuristic function on the grid decomposition and $cp \in \{p, e\}$ and indicates the current player that has turn at t.

Given this state-space representation the search problem is formulated as follows.

Initial state. The initial state is $s(0) = ((p(0), e(0), h(0)), p)$. The position of the pursuer after the end of the detection phase or directly the one supplied in input if the players have already line-of-sight.

Player. The pursuer plays at even rounds, whereas the evader during odd time steps. The first player that is allowed to move depends on the value of the time step at the end of the first phase.

Actions. Starting from a state $s(t) = ((p(t), e(t), h(t)), cp)$, the current player cp can move to any point $q' \in \mathscr{P}$ that is reachable given its speed and provided that there exists a safe path, that lies inside the polygon and does not collide with the contour of the map. After reaching the new point q', cp makes a new perception. The other player $\overline{cp} = \{p, e\} \setminus \{cp\}$ cannot make any action, but can still perceive.

Transition function. After the application of an action onto the current state, which can be described in words as "cp moves to q'", the result is a new state $s(t+1)$ where the position of the current player is updated to q' that triggers the update of the distance value. It might happen that, after a movement, line-of-sight is lost. If this is the case, the Minimax algorithm is interrupted and the pursuer has to fall back to the detection phase in order to discover again the evader's position before resuming the capture phase. Moreover, the granularity of the state-space representation is crucial, if for instance the size of the edge of the grid cells is too small, the resulting number of states is very high, and both branching factor and depth of the solutions become an issue, resulting in the problem becoming intractable. It should not get too coarse either otherwise the state-space representation would be inaccurate.

Terminal test. It checks whether one of the following conditions are satisfied: *(i)* $h(t) = 0$ in a state $s(t) = ((p(t), e(t), 0), cp)$, *(ii)* there is a loop in the path from the initial state to the current one, *(iii)* $t > T$, where T is the maximum allocated time for the game.

Utility. It is defined as $u(s(t))$ where $s(t)$ is a terminal state. If the pursuer wins (case *(i)*) the utility function returns the number of time steps required to reach $s(t)$, otherwise if the evader wins (cases *(ii)* or *(iii)*), $+\infty$ is returned.

A solution to this search problem is defined as the finite sequence of states $S = \langle s(0), s(1), \dots s(n) \rangle$, such that $s(0)$ is the initial state of the capture phase and $s(n)$ is the state that satisfies the terminal test. A solution S^* is optimal when it minimizes the number of times steps for the pursuer and at the same time it maximizes them for the evader, i.e., no player gains value by changing an action in the solution.

4.3.2 Finding a solution and speed ups

The algorithm is the standard Minimax one, with some enhancements. It begins with applying the terminal test to the initial state, if the outcome is true, the game ends. Otherwise, the actions and transition functions are called yielding the next states. This process is repeated recursively in a depth-first fashion until a leaf node is found. At this point, the utility function is called yielding the value for the player that gets propagated through the non-leaf nodes up to the root. Using this approach, other branches are expanded and evaluated, resulting to pruning if the value is worse than the candidate solution. Otherwise a replacement is performed and the currently evaluated branch becomes the new candidate solution.

As already stated in Section 2.4, the worst-case computational complexity of the Minimax algorithm is exponential with respect to the depth of the solution, where, in this context, it is the number of time steps needed to reach a terminal state. In order to keep the problem tractable, a number of techniques for speeding up the computation while preserving optimality are now introduced and discussed.

Actions within a time step. If the two players can see each other, actions leading to the furthest grid cells that are reachable within a time step are considered, according to the velocity of the players. Differently from considering just actions going to the adjacent grid cells, this reduces the depth of the branches that the search algorithm evaluates at the cost of slightly increasing the branching factor. The optimality is still preserved, as intuitively both pursuer and evader want to get closer or escape further, respectively.

Dominated actions. The evader discards the actions that would lead to the same position of the pursuer, or that can be reachable within a single time step by the pursuer itself. If no actions are available the evader doesn't make any movement, this does not affect optimality as [69] already explains.

Duplicated states. During execution, the already expanded states are added to a closed list. Every time a new state is ready to be evaluated, it is checked whether it is duplicated or not. A state $s(t) = ((p(t), e(t), h(t)), cp)$ can be safely discarded if $\exists s(t') = ((p(t'), e(t'), h(t'), cp)$ in the closed list, with $t' < t$ such that $p(t) = p(t')$, $e(t) = e(t')$ (therefore $h(t) = h(t')$). Likewise for dominated actions, [69] shows that the algorithm is still complete and optimal since the solutions that lead to duplicate states will have higher cost for the pursuer (or lower for the evader).

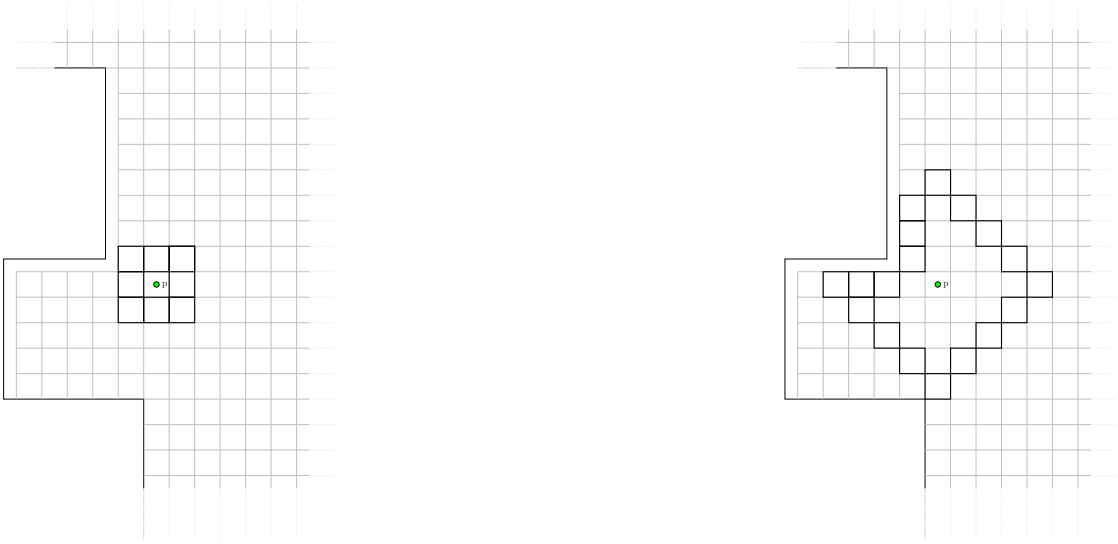


Figure 4.5: Neighbor cells with speed $k = \frac{\nu}{e} = 1$ **(a)** and with $k = 4$ **(b)**

Faster robots. As already discussed in Section 4.1.2, the usage of the grid can lead to a state granularity that might be too fine. This could dramatically affect the depth of the solutions. The successor states are the positions encoded into the center of the neighbors of the cell that contains the player which has turn. An interesting approach is to increase the speed of the robots and consider as next points the center of the cells that can be reached assuming such speed. Figure 4.5 shows the difference. The cells that are candidates to get selected as the next state of the game are highlighted using a ticker border. On the left hand side **(a)**, there's the situation in which $\nu = e$ whereas on **(b)** the speed is quadruplicated leading to $k = 4$. Notice that, given the matrix constructed using the heuristic function (which is depicted in Figure 4.1), it is straightforward to determine the furthest cells that satisfy the condition such that their distance is less than or equal to the allowed speed. This technique affects the progress of the underlying game and not the Minimax itself so the optimality condition is unchanged.

Alpha-beta pruning with heuristic. Every node in the search tree carries two values, α and β, representing the best values found so far in a branch for the pursuer and evader, respectively. The pruning condition for a particular branch is triggered whenever $\alpha \geq \beta$, namely when it is discovered that a branch leads to a solution that is worse or equal to a solution already found. Pearl [66] demonstrates that α-β pruning preserves optimality. In the best-case scenario, the worst-case computational complexity can be reduced to $O(b^{\sqrt{d}})$, where b and d are the branching factor and tree depth, respectively. However, the complexity highly depends on the order in which the successor states are expanded. The approach in this work is to sort new states in increasing order of heuristic value when pursuer has to move and vice versa for the evader. This is done in order to select as early as possible the "best" move from the point of view of the moving player according to a heuristic function $\hat{h}(t) = \frac{h(t)}{\nu}$, The heuristic value encoded in a state can then be used to estimate the depth of the solution. The idea is to imagine that the pursuer is always able to move at full speed while the evader remains on its position the whole

time. This estimate is added to the current time step t. If the result is greater than or equal to the time (cost) of the current best candidate solution, the branch is pruned. This operation is safe because this is a best-case scenario with respect to the pursuer (that is also unlikely to happen).

Lemma 1. *The estimate* $\hat{h}(t) = \frac{h(t)}{\nu}$ *is a lower bound on the optimal capture time* t^* *for the pursuer.*

Proof. Suppose that at time step t the pursuer makes an estimation and completes its turn by making a move. Now is the turn of the evader, 4 different situations regarding the distance $h(t)$ can occur:

(i) it stays the same therefore the best-case estimated depth of the solution from the point of view of the pursuer becomes $1 + \hat{h}(t+1) \geq \hat{h}(t)$,

(ii) it increases, leading to the same case as in the previous one,

(iii) it decreases and the pursuer manages to capture its adversary, however this cannot happen since the evader would discard this action,

(iv) it decreases and the pursuer is not able to win, since the evader manages to escape the depth of the solution is certainly higher than the one originally estimated.

As such, $\hat{h}(t)$ is a lower bound on the optimal capture time t^* for the pursuer. □

With such lemma, it is possible to prove that a branch can be safely pruned according to $\hat{h}(t)$.

Theorem 1. *The current evaluated branch can be pruned if its current cost* $g(s(t))$ *summed with to* $\hat{h}(t)$ *is greater or equal than the depth of the best candidate solution, preserving optimality.*

Proof. Suppose that a current evaluated branch with $g(s(t)) + \hat{h}(t)$ greater than or equal to the current cost of the best candidate solution is expanded. Given that $\hat{h}(t)$ is a lower bound according to Lemma 1, after some iterations, the value would be worse than the best solution found so far. As such, α-β pruning will discard that branch, proving the theorem. □

4.4 Phase 2: capturing the evader – Monte Carlo Tree Search

The second proposed algorithm for planning a motion strategy after establishing line-of-sight is Monte Carlo Tree Search. The space-state representation remains the same as in the Minimax case. One of the major differences is the criteria for traversing the search tree from the root node down to a particular leaf node which has to balance exploration and exploitation (more on Section 4.4.2). From a current leaf node a game is played out till the very end generating nodes at random (that's

why the words Monte Carlo are present). The outcome of the playout is propagated from the leaf node up to the root. When the timeout is reached, or the number of playouts executed reaches a certain limit, a move is selected considering the nodes that has the largest ratio of wins with respect to the player that has turn.

4.4.1 Using random sampling when making decisions

In certain instances of the problem, Minimax may have issues into finding a solution before the time allocated expires. That's when Monte Carlo Tree Search comes into place. The main part of the algorithm is shown in Algorithm 4. The procedure begins with the creation of the initial state, taking into account the position of the pursuer at the end of detection phase coupled with the time step, in order to decide who has turn first.

As in the Minimax case, a state-space representation is utilized, which is the same as defined in Section 4.3; actions and terminal state are also reutilized. The utility function is not considered in this context since the criteria for selecting the actual action to perform is different.

The outermost loop iterates as long as the global timer hasn't expired yet and the current state is not terminal. If both these conditions are satisfied, one iteration is made. It begins with setting up the necessary variables for running a single instance of MCTS. As long as there's time allocated or the number of simulations performed has not reached the threshold (which is a parameter supplied in input), the instance continues execution. One iteration consists of the implementation of the 4 phases depicted in Figure 2.4.

Selection. Before selecting a child node in the selection phase, the algorithm verifies that all of these children have at least one simulation in earlier iterations. The criterion for selecting a node is one of the key aspects of the algorithm and it is described in detail in Section 4.4.2.

Expansion. Then, the expansion phase is encountered, in which one child node is expanded. The idea is to prefer nodes that are most promising with respect to the player that has turn in the parent node according to the heuristic function that computes distance using the grid decomposition, much like the frontier ordering in the Minimax algorithm. This is used in order to exploit as much as possible the speedup in terms of time complexity in Alpha-Beta pruning.

Algorithm 4 Monte Carlo Tree Search algorithm

```
1: function MCTS(pursuerPos, initialTimeStep)
2:     initialState ← MCState(pursuerPos, evaderPos, initialTimeStep)
3:     rootNode ← MCNode(initialState, null, 0, 0)
4:     while hasTime ∧ rootNode.isTerminal() do
5:         iterations ← 0
6:         turnTimer.start()
7:         currentNode ← null
8:         while turnTimer.hasTime() ∧ iterations < maxIterations do
9:             currentNode ← rootNode
```

```
10:             while currentNode.isTerminal() ≠ true ∧ currentNode.isFullyExpanded()
    = true do
11:                 currentNode ← currentNode.select()                ▷ Selection
12:             end while
13:             if currentNode.isTerminal() ≠ true ∧ currentNode.isFullyExpanded()
    ≠ true then
14:                 currentNode ← currentNode.expand()                ▷ Expansion
15:             end if
16:             result ← currentNode.simulate(rootNode.type())        ▷ Simulation
17:             while currentNode ≠ null do
18:                 currentNode.propagate(result)
19:                 currentNode ← currentNode.parent()                ▷ Backpropagation
20:             end while
21:         end while
22:         rootNode ← rootNode.optimalChild()
23:     end while
24: end function
```

Simulation. Starting from the newly expanded node, an instance of the game is played. Algorithm 5 shows how this is performed. The simulation runs as long as the current depth is below an upper bound, called maxDepth that is supplied as an input parameter or the current node is not a terminal one. If both conditions apply, an iteration is performed. The procedure is pretty simple, it continuously selects random nodes going in depth as long it is possible. At the end of the while loop, the goal test is applied to the last node selected. The result value reflects the point of view of the player that has turn (rootType): if there's a loop in the path and the pursuer is playing it gets -1, 1 otherwise. The other way around is returned if the goal test is performed by the evader. Finally, if the maximum allowed depth is reached before a terminal node is found, nothing can be said about the outcome of the playout, therefore a value of 0 is returned.

Algorithm 5 Playout simulation algorithm

```
1: function SIMULATE(rootType)
2:     currentNode ← this
3:     currentDepth ← 0
4:     while currentDepth < maxDepth ∧ currentNode.isTerminal() ≠ true do
5:         currentNode ← currentNode.randomSelect()
6:         currentDepth ← currentDepth +1
7:     end while
8:     if currentNode.goalTest() = LOOP then
9:         return rootType = PURSUER ? -1 : 1
10:    else if currentNode.goalTest() = CAPTURE then
11:        return rootType = PURSUER ? 1 : -1
12:    else
13:        return 0
14:    end if
15: end function
```

Another crucial part is how to actually perform the random sampling, Algorithm 6 carries out this task. In particular, it uses a geometric distribution which has a probability mass function of $(1-p)^k \cdot p$, where k is the number of independent trials, each with success probability p, a that can be seen as a discretized negative exponential distribution, to select the next state. The *support set* which is all the possible values that k can assume, is mapped to the indexes of nextStates that contains the next candidate states. Moreover, the distribution is simple to characterize: only the probability of selecting the first element of the support, which in this formulation corresponds to the first element of nextStates, is necessary. Before making a choice, the successor function is called returning the set of states that are candidate for selection. As usual, the order in which states are put in the list depends on the value of the heuristic function with respect to the player that needs to randomly selects its next action, having the most favorable one positioned at the front.

The random distribution is setup so to have a probability to select the most promising state equal to $p = 1/3$. The rest of the cumulative probability is distributed in a decreasing order determined by the probability mass function among all the other states.

After all this setup phase, the real random sampling takes places. A number is drawn from the geometric distribution until it can be utilized as a valid index in the nextStates list. Therefore at the end, the state that is in matching position within this list is returned to the caller.

Algorithm 6 Selecting a state using random sampling from a geometrical distribution

```
function randomSelect
    randomGenerator ← RandomGenerator(randomDevice)
    randomDistribution ← GeometricDistribution(1/3)
    nextStates ← SuccessorFunction()
    number ← nextStates.size()
    while number ≥ nextStates.size() do
        number ← randomDistribution.drawNumber(randomGenerator)
    end while
    return nextStates[number]
end function
```

Backpropagation. Once the simulation phase is completed, the last step, called backpropagation is performed. The result value computed by Simulate (Algorithm 5) is propagated from the leaf node up the root. In particular, for node i we have that $R_i = \sum_{k=0}^{v_i} R_k$ where v_i is the number of visits of the i-th node. In words, an algebraic sum of the results computed among all the simulation along the branch being selected is performed. In this phase, also the number of visits of the nodes along the path up to the root is increased by one.

After an instance of MCTS is completed, the optimal child is selected, the criteria utilized is to choose the node that has the most number of wins R_i with respect to the player that has turn. After that, another iteration of the outermost loop is performed returning to the initial situation.

4.4.2 The exploration vs. exploitation dilemma

One of the major difficulties in selecting child nodes is to maintain a fair balance between *exploitation* of deep nodes exposing a high value of win rate and *exploration* of moves with low amount of simulations.

Chaslot [20] shows a number of formulas to compute the selection of the most promising child nodes.

UCT guarantees a good balance between complexity and performance. Interestingly the formula is one of the first that has been designed for this kind of problems. It is based from the UCB1 formula of the Multi-Armed Bandit theory [46]. The approach is to select the child node that maximizes the quantity

$$\frac{R_i}{n_i} + C\sqrt{\frac{\ln n_p}{n_i}}$$

where:

- R_i is the algebraic summation of the outcomes of previous playouts that start either from node i or one of its descendants,
- n_i is the number of visits performed on node i,
- C is a tunable exploration parameter, by default equals to $\sqrt{2}$ but in general it is selected empirically based on the problem,
- n_p is the number of visits of i's parent node p.

By combining the UCT formula with simulations, depending on how much time is allocated, the MCTS estimation will be more and more accurate. Ideally, if the time had been infinite, the accuracy would be perfect, thus effectively resulting in MCTS converging towards Minimax.

Chapter 5

Implementation

This chapter presents the general architecture of the system, in terms of modules implemented for performing the research study.

Section 5.1 shows the overall architecture of the simulation software extensively utilized for the experimental evaluation. Section 5.2 presents all the programming languages and technologies used to develop such software. Section 5.3 describes some enhancements I implemented. These techniques extend the capabilities of the proposed algorithm, by allowing a more efficient finding of a solution.

5.1 The simulation software

The functionality of the application can be expressed in terms of several requirements:

$\mathcal{R}_1$ The application should encourage modularity by enforcing a logical separation through a set of different modules interconnected between each other. The software is developed from scratch using several frameworks, and should supports two different user interfaces: one using a Graphical User Interface (GUI), and one utilizing the Command Line Interface (CLI).

$\mathcal{R}_2$ The application allows a user to load an environment file, with an ad hoc format, which is explained in Section 5.1.4.

$\mathcal{R}_3$ Each user interface exposes a set of actions that directly influences the parameters and input data utilized in the algorithm such as the size of the edges of the grid decomposition or the amount of discretization points for the inducing segments of the visibility-based decomposition. In the graphical version this is usually carried out using menus and several layers of sub-menus and actions, whereas in the textual interface, the set of options are supplied as input when the executable file is invoked from the command line interpreter.

$\mathcal{R}_4$ After the initialization phase is completed, it is possible to start the execution of the algorithm. As it computes the strategy, the current search progress can be visualized either on the GUI as current paths of the players (see Figure 5.2), or by manually reading the printed messages in the command line window that report the movements of the players (see Figure 5.3).

$\mathcal{R}_5$ The search progress is also logged into a file that can be opened at later time to analyze a previously completed instance.

The software implements precisely the formal definition of the problem given in Chapter 3.

Starting from the requirements specification, the chosen architectural structure consists of three modules: the core module that comprises all the algorithms and the data processed by them, the Graphical User Interface (GUI) module that provides a rich set of options for the user who interacts with the software, and finally a Command Line Interface version that requires less dependencies and it is more suitable to automatize processes such as the several experiments used to evaluate the research conducted in this thesis.

5.1.1 The core module

This module implements all the logic paired with the necessary data in order to execute the proposed techniques formalized in the previous chapters. The main idea that led to separate this component from the rest is to increase as much as possible the reusability, emphasizing on the concept of task separation: every module should carry out a tight software requirement. Put in practice, this module exposes a set of APIs that represent the "access door" through which other modules should pass in order to interact with this one. A set of precise inputs and outputs are also defined so to allow a clearer and standardized design of external modules regarding the I/O layer.

5.1.2 GUI module

One of the two ways for running a simulation is to use the Graphical User Interface. Figure 5.1 shows the main window along with a sample map loaded. There are several menus that can be used to set all the necessary parameters and the polygonal decomposition required to carry out the search inside the environment. The window provides several menus that encapsulate many actions that allow a high degree of interaction with users, in particular the «Options» menu that contains several commands to configure the parameters of the algorithms.

A typical usage work flow is the following:

- the user loads a map by accessing the «File» menu and then clicking on «Open»,
- after loading a map, the polygonal partitions must be created, by clicking «Pair Vertices Partition» and «Grid Partition». Note that is mandatory to create both. Regarding the Pair Vertices Partition, the user must input the number of discretization points for each inducing line of the visibility-based decomposition, whereas for the Grid Partition, the size of the edge must be provided,
- after that, the user can select the type of algorithm that is used for the second phase of the game, namely either Minimax or MCTS,

Figure 5.1: Main screen of the graphical simulation application.

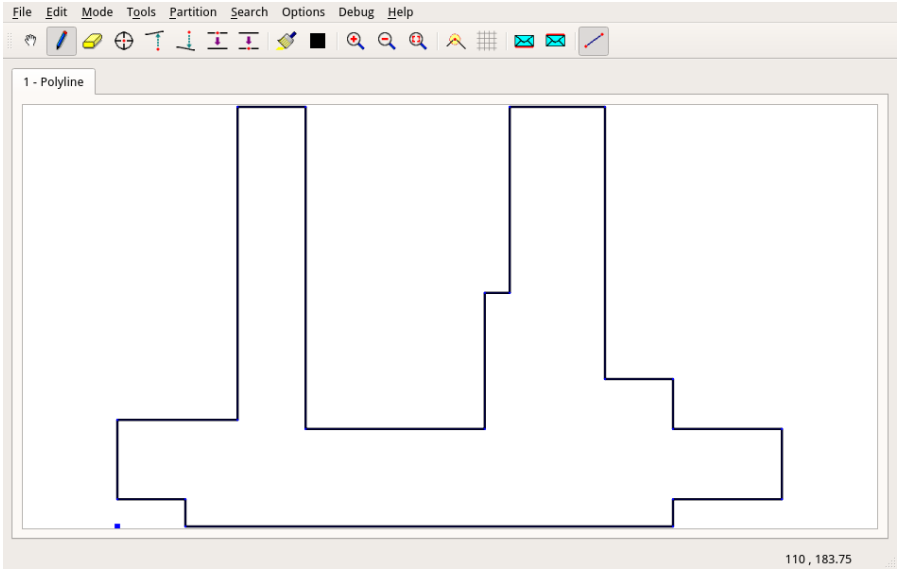
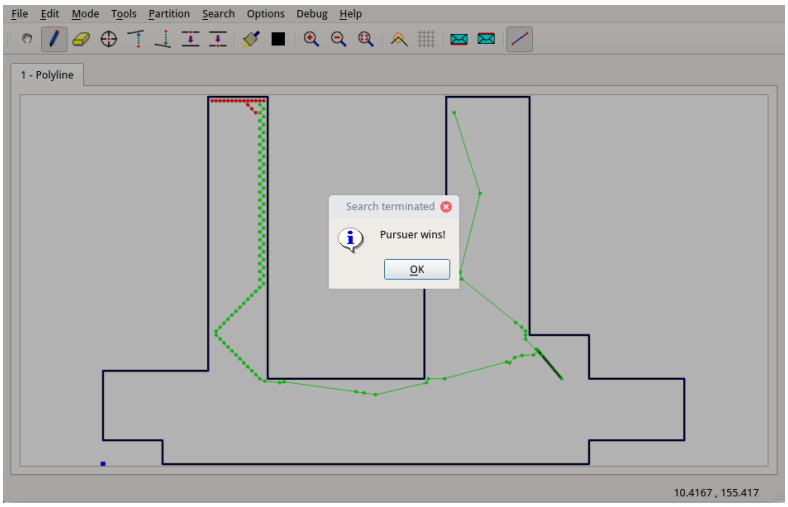


Figure 5.2: Example of a completed simulation with the pursuer as a winner.



- the search can be initialized, by clicking on menu «Search» and then «Initialize Search», the x and y coordinates of both pursuer and evader are asked in input using dedicated dialogs, along with their speed,
- finally, by clicking «Search» then «Start Search» the simulation can be started. During the first phase, the movements of the pursuer are displayed while it moves in the environment to clear it. In the second part, after Minimax or MCTS completes its execution, the optimal strategy is drawn onto the map.

At this point the user has to wait until the algorithm completes its execution. If the pursuer wins, an informative dialog appears telling the user that it has won the game as depicted in Figure 5.2. Any other conditions, already discussed in Section 3.1, will display a window informing the user that the evader has won.

Being a multithreaded application, during execution of the search procedure, the GUI is still responsive. Therefore, if necessary, users can stop the algorithm, by

Figure 5.3: Example of a partial simulation using the CLI application.

```
raffaele@arch-Lenovo  cmake-build-release  feature/fork  ./search/search --map_file ../data/poluline/albert_b_laser4.dat --cell_size 2 --discretization_points 5 --pursuer_x_pos 63 --pursuer_y_pos 2 --evader_x_pos 240 --evader_y_pos 108 --algorithm=G --robots_speed=1
NUMBER OF CELLS: 1822
DONE
FREE CELLS: 14647, 7683
E1127 22:03:33.921545 25950 search.cpp:122] PARAMETERS: algorithm=G; mov_cost=1; robots_speed=1; interval=10000; cell_size=2; discretization_points=5
E1127 22:03:33.937639 25950 node.cpp:111] Node depth=0 g=0 state=((63/1 2/1),(240/1 108/1)), hp=0, he=0, hc=151.660
E1127 22:03:33.971177 25959 search.cpp:386] TIME STEP: 1, MOVING FROM 63/1 2/1 TO 920/13 350/13
E1127 22:03:34.004238 25959 search.cpp:386] TIME STEP: 2, MOVING FROM 920/13 350/13 TO 66/1 38/1
E1127 22:03:34.038096 25959 search.cpp:386] TIME STEP: 3, MOVING FROM 66/1 38/1 TO 6498/103 4452/103
E1127 22:03:34.070504 25959 search.cpp:386] TIME STEP: 4, MOVING FROM 6498/103 4452/103 TO 69120/1043 51000/1043
E1127 22:03:34.101552 25959 search.cpp:386] TIME STEP: 5, MOVING FROM 69120/1043 51000/1043 TO 660015/9896 556575/9896
E1127 22:03:34.133200 25959 search.cpp:386] TIME STEP: 6, MOVING FROM 660015/9896 556575/9896 TO 51700/787 45974/787
E1127 22:03:34.155844 25959 search.cpp:386] TIME STEP: 7, MOVING FROM 51700/787 45974/787 TO 63/1 61/1
E1127 22:03:34.177865 25959 search.cpp:386] TIME STEP: 8, MOVING FROM 63/1 61/1 TO 31548/497 156384/2485
E1127 22:03:34.210345 25959 search.cpp:386] TIME STEP: 9, MOVING FROM 31548/497 156384/2485 TO 19725/311 99731/1555
E1127 22:03:34.232204 25959 search.cpp:386] TIME STEP: 10, MOVING FROM 19725/311 99731/1555 TO 376830/5903 409965/5903
E1127 22:03:34.265666 25959 search.cpp:386] TIME STEP: 11, MOVING FROM 376830/5903 409965/5903 TO 65/1 70/1
E1127 22:03:34.290004 25959 search.cpp:386] TIME STEP: 12, MOVING FROM 65/1 70/1 TO 16782/251 17593/251
E1127 22:03:34.317139 25959 search.cpp:386] TIME STEP: 13, MOVING FROM 16782/251 17593/251 TO 69/1 73/1
E1127 22:03:34.341791 25959 search.cpp:386] TIME STEP: 14, MOVING FROM 69/1 73/1 TO 68631/958 72279/958
E1127 22:03:34.368255 25959 search.cpp:386] TIME STEP: 15, MOVING FROM 68631/958 72279/958 TO 71/1 77/1
E1127 22:03:34.400313 25959 search.cpp:386] TIME STEP: 16, MOVING FROM 71/1 77/1 TO 73/1 77/1
E1127 22:03:34.425030 25959 search.cpp:386] TIME STEP: 17, MOVING FROM 73/1 77/1 TO 773591/9852 717155/9852
E1127 22:03:34.450310 25959 search.cpp:386] TIME STEP: 18, MOVING FROM 773591/9852 717155/9852 TO 742271/8617 684095/8617
E1127 22:03:34.476682 25959 search.cpp:386] TIME STEP: 19, MOVING FROM 742271/8617 684095/8617 TO 1314843/14909 1190939/14909
E1127 22:03:34.507540 25959 search.cpp:386] TIME STEP: 20, MOVING FROM 1314843/14909 1190939/14909 TO 91/1 77/1
E1127 22:03:34.530089 25959 search.cpp:386] TIME STEP: 21, MOVING FROM 91/1 77/1 TO 7513/83 6598/83
E1127 22:03:34.560520 25959 search.cpp:386] TIME STEP: 22, MOVING FROM 7513/83 6598/83 TO 4701/52 16787/208
E1127 22:03:34.584477 25959 search.cpp:386] TIME STEP: 23, MOVING FROM 4701/52 16787/208 TO 89/1 83/1
E1127 22:03:34.608233 25959 search.cpp:386] TIME STEP: 24, MOVING FROM 89/1 83/1 TO 9751/109 9166/109
E1127 22:03:34.633551 25959 search.cpp:386] TIME STEP: 25, MOVING FROM 9751/109 9166/109 TO 87/1 87/1
E1127 22:03:34.658612 25959 search.cpp:386] TIME STEP: 26, MOVING FROM 87/1 87/1 TO 85/1 89/1
```

clicking on «Stop Search» in the «Search» menu. In this case it is assumed that the evader wins.

As already mentioned, another functionality is the possibility to open a previously completed search using the log generated using glog, and to display the sequence of movements performed by the players along the map.

5.1.3 CLI module

Another possibility for executing the searches is to utilize the Command Line Interface. Thanks to the modularity of the system, it is straightforward to combine the core package that contains the algorithms and a particular interface that allows user interaction.

The set of parameters has to be provided using the appropriate names and their legal values. This is implemented using the gflags library [33]. It takes care of all the parsing and types conversions, providing a simple interface for defining and accessing the flags throughout the code. Figure 5.3 shows an example invocation of such application along with some parameters, specified as input. Note that a subset of parameters is mandatory to be specified.

The motivation behind the implementation of a second user interface, comes from the necessity of more automation to execute the experimental evaluation, that a GUI fails to comply. In fact, as the reader can notice in Figure 5.3, the number of parameters is usually quite large and therefore this application is not very well suited for manual utilization.

5.1.4 Creating the environments

The maps used for the experimental evaluation are taken from the Radish repository [34], which provides several maps encoded as log files that record all the measurements done by a robot equipped with a laser sensor. The whole process of controlling the robot, collecting the data from its sensors, and eventually generating an image of the map is carried out by a set of modules and programs that are part of CARMEN [19]. Note that this is not true for all maps present in the repository. This thesis uses only maps that are compatible with CARMEN. In particular, the

image generation tool that creates a PNG file from a previously collected log is used. The application allows a fine tuning of the resolution and dimensions of the image to process as well as various controls to reduce the noise that comes from the sensors.

The simulation program implemented in this thesis expects a map to be encoded into a sequence of control points that describes a polyline. Therefore, starting from a raster image, a couple of steps have to be performed in order to produce a file that adheres to the format.

First, the Python script shown in Appendix B.1 is executed. It takes in input the image file and performs a walk along the border. Whenever a change of direction is detected, a control point is added to a list. The frame of reference is the same of the PNG image. . For the inner walls a recursive approach is used. The algorithm stops whenever it crosses the initial point, thereby closing the path. At the end, the program saves the resulting list in a file having the format expected by the simulation program. The second Python script shown in Appendix B.2 applies a scaling factor to the coordinates. This is used for adjusting the scale of the map. Section 6.1.1 describes in detail how this is accomplished.

5.2 Frameworks and technologies employed

This section gives an overview of the relevant basic blocks that have been used to build the final application. First in Section 5.2.1 I will discuss the programming language that has been selected for implementing the software. Then each subsequent section (namely Sections 5.2.2, 5.2.3, and 5.2.4) presents a specific software library written in the same language. Each choice will be motivated, highlighting the relevant modules of such frameworks effectively utilized in the application.

5.2.1 Programming languages and paradigms

The programming language selected for the implementation is C++, which is an object oriented language that extends the capabilities of C.

One of the drawbacks of C is that the programmer is forced to design and implement software using the procedural paradigm, which with large and complex code bases doesn't work quite well. The object oriented paradigm overcomes these issues by introducing the concept of *objects* modeled using classes. Each *class* allows a set of operations called *method* which are functions just like in the procedural paradigm. These methods usually perform some kind of computation on variables called *fields*.

An important concept is *information hiding*, i.e, the possibility to define the visibility of methods and fields, only exposing to public the relevant parts of functionality that a programmer wishes to. Another key idea is *inheritance* which allows the definition of a general class that provides some kind of functionality that is later extended with a subclass that has the same functionality augmented with more specific behavior depending on the concept being modeled.

The language comes with a standard library called Standard Template Library (STL) which implements a lot of different data structures and algorithms that are

frequently used in many applications. For instance the class `list` models the data structure of a double linked list of objects, the same class is utilized in different ways each time containing a specific type of object thanks to the generality offered by the template mechanism. Another container used is `vector` that has a similar behavior of classic arrays with cells that contain values allocated in RAM contiguously.

Another feature inherited from C is the possibility to manage dynamic memory by hand through pointers. The main advantage is the ability to have precise control of RAM usage and resource consumption in general but the drawback is that the programmer needs to track precisely every allocation and plan the deallocations wisely. If a deallocation is performed at wrong time, the code might try to access an object that is no longer in memory and therefore a program error occurs. It might happen that the programmer forgot completely and the portion of memory becomes unusable as long as the program is executing.

The issue of tracking down all these things becomes more and more tedious as the complexity of the project increases, that's why the STL provides classes for implementing smart pointers which encapsulate the original pointer along with reference counting. Every time a new variable owns this pointer, the counter gets increased by one, if there are no variables, the counter reaches value zero and the memory region pointed by the underlying pointer gets deallocated automatically. The classes that implement this behavior are `shared_ptr` and `unique_ptr`, that respectively allow multiple and single owners of the underlying memory region in a given instant time.

At last, as the implemented software in this thesis uses threads, care must be taken in order not to modify the content of variables simultaneously from different threads. Since the order of execution is scheduled by the underlying operating system the order of modification cannot be predicted in advance and therefore undefined behaviors might occur. Such undefined behaviors are called *race conditions*. To prevent them, the code that accesses these variables must be surrounded with special objects called *mutexes* that allow the entrance of the region of the code, called *critical section* to a finite number of threads, usually only one. This can be done explicitly by the programmer, but if the critical section consists of modifying a primitive type variable, the programmer can resort to the class `atomic<T>` which ensures thread safe load and store of a value. This is used for the flags that signal the expiration of the global timeout timer and of every turn of MCTS.

5.2.2 CGAL

As mentioned in Section 2.6 and in Chapter 4, computational geometry plays a big part in providing the necessary algorithms and data structures that help to solve the pursuit-evasion problem. In particular the library used is called CGAL [78] and is developed using the same approach of massive usage of template programming like in the STL. The framework is composed of many modules. The ones used in this application are:

- 2D Arrangements, an **Arrangement** allows the possibility to add a set of planar curves that induce vertices, edges, and faces. The module also allows

the possibility to make queries in order to discover the geometrical entity that resides in the point being queried. They are used to encode the map and its grid and visibility-based partitions.

- 2D Visibility, starting from an arrangement and a query point, the visibility area is computed using dedicated and efficient algorithms. This is a crucial when computing the visibility region of the players.

- 2D Regularized Boolean Set-Operations implements the common set operations like, union, difference, and intersection applied to polygons, this is used heavily by the algorithm that deals with the first phase of searching.

- Qt integration, it implements a set of widgets that can be added to a regular window and display the geometric objects defined in other modules, for instance in the graphical application the main widgets allow the display of the 2D arrangements.

Unfortunately, some modules are not completely interoperable – e.g., 2D arrangaments and Boolean Set Operations. Therefore, some custom methods have been implemented to convert an object from one type to another when it is needed, without losing information.

5.2.3 Qt

Qt [68] is an open source cross platform framework that encompasses a lot of different modules. It is used primarily for its powerful and flexible graphical toolkit that allows a rapid development of GUIs. It also provides a threading library available many years before STL library was released. Since Qt is already used for the GUI, it is a reasonable choice to use its threading library as well.

As explained in the previous section, CGAL provides a dedicated module that operates nicely with Qt allowing a tight integration between these two frameworks and the possibility of developing more eye-pleasing and richer user experience with respect to a command line implementation.

A central and powerful feature provided by the Qt framework is the signals and slots mechanism. Such a mechanism allows communication between objects. It is basically an alternative to the callback technique. When a particular event occurs, a signal is raised, if there is a slot connected to that signal, it gets invoked reacting to that event performing some computation. This patterns plays nicely with user input on a GUI: if a button is pressed, it raises the signal `clicked()`. Now suppose that there's a slot called `showAlert()` connected to the `clicked()` signal. Once a user actually presses that button, the slot is invoked and a nice message window is displayed.

This mechanism is not only used for GUIs but also in the threading library, whenever a thread has just started or completed, or in the networking library, for instance when a socket acknowledges the fact that there is new data ready to be processed sent from a remote endpoint and raises the appropriate signal so that the rest of the application is aware of this event. Programmers can also define custom signals in implemented classes. The simulation software takes advantage of this.

For instance when the search algorithm completes, it raises a signal of completion that is caught by the main window that displays an informative message on screen.

The cross platform feature allows the possibility to write code only once and then recompile it for different operating systems and hardwares supported.

5.2.4 Boost

The choice of Boost [14] is somewhat mandatory since it is a dependence for using CGAL. It is a modular, cross-platform, open source framework. Besides the internal usage in the computational geometry library, the module for building hashing function is used since some hash tables are employed to speedup the computation, as explained in Section 5.3. The complex types used as the key of these hash table are constituted of primitive types that have already a proper hashing function defined in the STL. Boost provides a handy method that combines the result of previous hash functions in order to come up with a robust implementation compared to a hand made one.

5.3 Code tweaks

Besides focusing on an efficient implementation of an algorithm *per se*, sometimes there are bottlenecks that rise in the accompanying data structures and algorithms. A typical approach is to use Valgrind [80], which is a tool to detect not only such issues, but also for debugging and profiling memory leaks and other related problems.

Heuristic function caching. The tool highlights the fact that the computation of the heuristic function, which is heavily used across all three algorithms illustrated in this work, can be slow, especially in environments with realistic dimensions. That's why a cache is employed to reutilize the previously computed matrices and save time. Of course, during the search, a lot of these matrices could be potentially created and since the amount of usable memory is finite, at some point some of them must be replaced with the new ones that are about to be used. The policy utilized is a classical Least Recently Used [73]. The idea behind this approach is to maintain a linked list of objects. Every time an object is accessed it is moved in front of the list. As a result, the most frequently utilized matrices are at the beginning of this list, whereas the least used (as the name of the policy itself suggests) are in the back. The list allows a fixed number of elements that can be present at a given instant of time. If it is full the replacing procedure takes place: the last item is discarded and deallocated. As a result, the empty spot in memory is filled with the newly created item which, according to the policy, it is moved at the front of the list itself. Besides this data structure, the core object is an hash table that uses as a key the position of the evader. This is because in general, it is the player that makes less movements, and through direct inspection of experiments, tends to move and evaluate a restricted set of successor states depending on its starting position in the simulation. Since the coordinates are complex aggregate types, a dedicated hash function has been implemented. Fortunately they are built upon integers and

therefore the standard library already provides a robust implementation of a hash function for such a type. The final piece is a method to combine all the results and to return a single number that is consumed by the hash table itself: this is carried out by Boost as already mentioned in Section 5.2.4. With this technique, a significant gaining in performance has been achieved.

Closed lists. When it comes to algorithms that are applied to a search-state paradigm, sooner or later they will end up evaluating some already encountered states. This is when closed lists are very useful: they maintain a list of previously expanded states and whenever a new one is ready to be evaluated, the algorithm checks whenever it is already present in the closed list or not. If this is the case the state could be discarded in order to save computational time and space. Some additional criteria could be considered before discarding, for instance, in Section 4.3.1, the duplicated states approach is enforced by checking the depth of the elements retrieved from the closed list. The closed list can be seen as a sort of caching mechanism, thus, likewise for the heuristic function, another hash table is utilized, this time using the state as the key. Despite its complex structure, it can be dissected with a combination of much simpler types that have already a hash function implemented, therefore the hash combining pattern using Boost is utilized again.

Parallelism. In the era of multi-core processors, the choice of parallelism is a natural consequence. Unfortunately it is not easy to apply to such complex algorithms maintaining consistency and stability. Moreover, the Minimax algorithm uses a pure depth-first approach that doesn't expand subtrees in parallel. Nevertheless, some degree of parallelism is used. In particular the algorithm that computes the grid decomposition divides the set of rows of the grid, such that up to 4 different threads process an independent range of rows.

Chapter 6

Experiments and results

This chapter wraps up all the procedures performed, and the subsequent data collected, in order to assess the performance and correctness of the methods that previous chapters define and implement. Section 6.1 begins with an overview of the setup and input data that experiments use, the selection and creation of the environments, in which way the starting points of the players are chosen, the values of the parameters across various executions, and the criteria used for evaluating the results.

The remaining part of this chapter is structured as follows: Section 6.2 analyzes the algorithm that clears the environment depending on how much discretization points are used for the inducing segments. Section 6.3 deals with an in-depth analysis of the Minimax implementation by comparing the results obtained by first setting the speed equal to the cells' edge size and then by increasing it. Section 6.4 performs a comparison between Minimax and MCTS, highlighting the differences and similarities, along with their corresponding strengths and weaknesses.

6.1 Experimental setup

This section summarizes the environments selected alongside with their characteristics. Then, the most important parameters of the proposed approach are described focusing on the values that they assume across all the experiments that are evaluated.

The simulations are performed over a set of remote Virtual Private Servers (VPS) all equipped with a 2 Virtualized Cores of a 3.20 GHz i7-3930K processor with 6 GB of RAM running Ubuntu 16.04 LTS.

6.1.1 Environments

The experiments consist of simulations over a set of maps. The source of these environments is the Radish repository [34]. Two of them are used in a prior work [69]. In particular, the authors select one map (*albert-b-laser*) with a single corridor and several simple rooms (Figure 6.1) and another one (*utk-claxton*) with a higher number of long corridors and a couple of rooms (Figure 6.2). This thesis further introduces two other maps: one (*fr079*) with a single long corridor but more

complex structure of rooms, such that some of them are reachable only through an anteroom that is accessible from the corridor (Figure 6.3), and another one (*ubremen-cartesium*) with a simpler structure but characterized by a large open space among with some rooms (Figure 6.4).

Figure 6.1: Environment *albert-b-laser*.

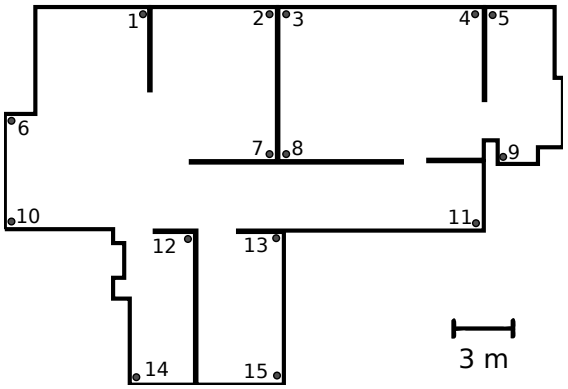


Figure 6.2: Environment *utk-claxton*.

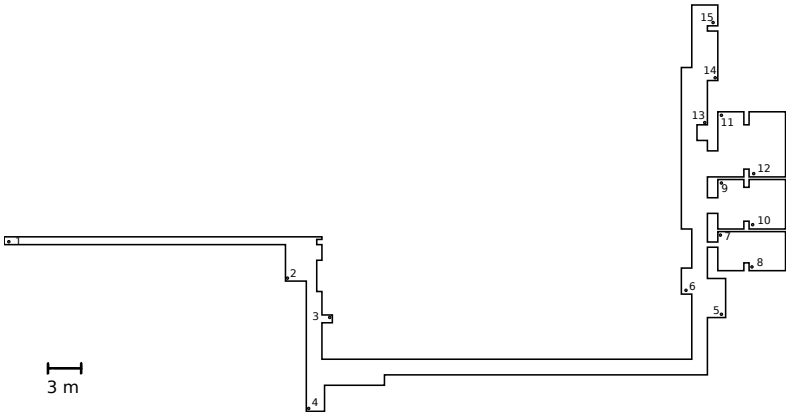
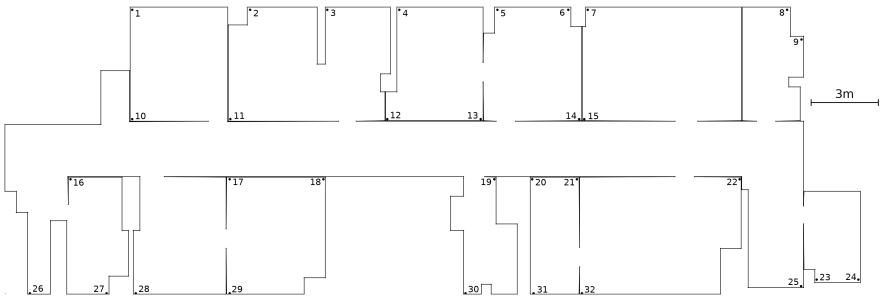


Figure 6.3: Environment *fr079*.



Note that the datasets do not report the scale of the original images. Therefore, a manual inspection is performed to convert the coordinates to match the real world distances.

Figure 6.4: Environment *ubremen-cartesium*.

Table 6.1: Environments with their sizes.

Map	Width (m)	Height (m)
albert-b-laser	30	24
utk-claxton	140	80
fr079	38	13
ubremen-cartesium	41	11

This ensures that simulations are done like as they are performed in the real environment providing more realistic results. This allows the derivation and discussion of much more useful insights about the validity of the methods being presented.

The approach to scale the maps is to measure the width in pixels of a doorway assuming that it corresponds to about 1 m. The resulting ratio m/pixels is applied directly to the coordinates effectively scaling the map by the desired amount. This is performed by the Python script illustrated in Appendix B. It takes as input a map file, that contains in textual format the set of control points that describes the closed polygonal line representing the environment, and a scaling factor and performs the necessary multiplications. At the end, it generates another map file with the correct data.

The line segment in each figure measures 3 m and the sizes of the environments are reported in Table 6.1.

6.1.2 Parameters

The number of parameters that need to be determined is relatively high. Some of them are in common across the settings and environments, whereas others are only valid either for the Minimax algorithm or for Monte Carlo Tree Search.

Timeout. The timeout is set to 5 hours.

Discretization points. The number of discretization points is reported in Table 6.2. Section 6.2 compares the performance of the detection algorithm with respect to the number of discretization points presented in the columns of Table 6.2. This is done in order to select the discretization that guarantees the most a fair distribution of points along the inducing lines, minimizing the problem of indistinguishability with respect to the heuristic function if the points are too close. Table 6.3 (column "Number of visibility cells") summarizes the number of

Table 6.2: Number of discretization points across different runs.

Map	1st group	2nd group	3rd group	4th group
albert-b-laser	5	10	15	20
utk-claxton	15	20	25	30
fr079	5	10	15	20
ubremen-cartesium	5	10	15	20

Table 6.3: Number of visibility and grid cells for each map.

Map	Number of visibility cells	Number of grid cells
albert-b-laser	1822	22661
utk-claxton	3498	34535
fr079	17950	70000
ubremen-cartesium	48669	96118

visibility-cells generated after the partitioning algorithm completes its execution. The significantly higher number of cells in *ubremen-cartesium* with respect to other environments is justified by the fact that there exists an open space in the middle of the map. Note that the number of discretization points is the same regardless of the length of the inducing line.

Cell size. Table 6.3 (column "Number of grid cells") shows the amount of the cells for each map. The number is a function of the scale of the environment, but ultimately all grids have cells with edge size $e = 0.10\,\text{m}$. This ensures that the state space is fine grained enough to meaningfully describe the environment and, at the same time, to not generate problems during the computation of Minimax and MCTS: that is the well known tractability issue introduced in Section 2.4, that this thesis addresses in Chapter 4.

Figure 6.5 summarizes the comparison between the number of visibility and grid cells for each map. In all environments the visibility-based decomposition manages to generate less cells, effectively compressing the state space and speeding up the computation.

Speed of the robots. Similarly to the selection of cell size, the choice of the speed is a function of the size of the map being considered. Three maps, *albert-b-laser*, *fr079*, and *ubremen-cartesium*, have comparable width and height and, therefore, the speed is the same. In *utk-claxton* we have longer corridors, therefore the speed is doubled in order to traverse them in a reasonable time. Table 6.4 wraps up the choice of the parameters.

MCTS timeout. The parameter is set to 45 s in the low timeout scenario for all the maps, and augmented to 90 s in the high timeout case.

MCTS maximum simulations depth. The value of this parameter is set online during the execution of the algorithm. It equals the value of $\hat{h}(t) = \frac{h(t)}{\nu}$, where $h(t)$

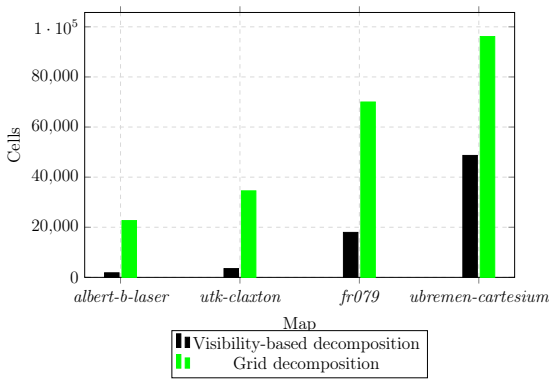


Figure 6.5: Cells comparison histogram.

Table 6.4: Speed of the robots across all the maps.

Map	Speed (m/s)
albert-b-laser	0.5
utk-claxton	1
fr079	0.5
ubremen-cartesium	0.5

is the distance of the players in the state $s(t)$ when line-of-sight is established for the first time. Likewise for the MCTS timeout, this parameter is doubled in the higher maximum simulation depth scenario.

The immediate result of augmenting both timeout and maximum simulation depth ensures that the algorithm is capable to compute a solution that closely resembles the one provided by Minimax in the same setting because, in general, the higher the number of simulations performed the better the move selected.

6.1.3 Batch of simulated searches

An environment, the set of all parameters correctly assigned, and the combination of the initial positions of the two players, represent a *setting*. A *batch* of simulations is composed of all the settings that correspond to all the possible valid combinations of the initial points.

For each map file, there exists an experiment file that contains the parameters that are in common to each setting, such as the number of discretization points, cell size, and the global timeout.

After that, the experiment file contains a set of initial points for evader and pursuer. The class `BatchExperimentDriver` iterates over these points by combining them, creating the actual setting ready to be executed. Of course, the settings where both pursuer and evader start from the same point are discarded. The choice of utilizing a remote machine is justified by the fact that, even with a relatively low number of points, the total amount of combinations can easily reach several hundreds. Since a setting might last some minutes, and sometimes a few hours, the

Table 6.5: Number of points selected along with the total amount of settings ($points \cdot (points - 1)$)

Map	Points selected	Total number of settings
albert-b-laser	15	210
utk-claxton	15	210
fr079	32	992
ubremen-cartesium	18	306

total execution time could be a couple of days long or even more. This requires an adequate uptime and reliability given that the process is entirely automatic and puts under stress the hardware for long periods of time.

The points have been selected in order to benchmark the algorithms under different conditions, for example the majority of points are placed inside the rooms. Some of them are placed in a corridor, or in an open space, because there is the possibility that the line-of-sight condition is verified when the distance of the players is still relatively high, resulting in a lot of options for the evader to escape and hide again.

Table 6.5 lists the amount of points selected for each map, and the number of combinations derived by pairing such points. In *fr079* more points have been selected because it has a higher complexity with respect to other environments. Overall, considering all the combinations of parameters, pairs of starting points, and algorithms utilized to compute a strategy, the total number of simulation runs made is about $10,000$. Note that for each setting, only one run for each algorithm is performed.

6.1.4 Metrics used for evaluation

For each instance executed, the number of time steps normalized with the step cost and the time needed to compute the solutions are measured. As Appendix A explains, the class `BatchExperimentDriver` collects these values from the instance that has just completed and saves them in a row of a dedicated Comma Separated Values (CSV) file. Using these values it is possible to compute both the average (μ) and standard deviation (σ) of number of time steps and of computing time. These two values should give meaningful insights about the performance of the methods being evaluated depending on the environment being utilized and the points being selected.

6.2 Clearing the environment with different discretization points

This first set of experiments assess the performance of the clearing algorithm with respect to different number of discretization points. Table 6.2 shows the values of the parameters for each map. All the instances terminate within the allocated timeout and the pursuer manages to detect the evader. Besides the metrics used and described in Section 6.1.4, an additional counter is introduced that measures

the number of times that the grid partition is used when the indistinguishability condition applies.

Figure 6.6 shows the average cost and standard deviation for each map and execution (run). As the reader can see, given a map, the values are more or less the same. The slight variations depend on the fact that a particular number of discretization points could potentially trigger some indistinguishability cases that may not happen with another value of the parameter, because the distribution of the points among each segment is different, . Figure 6.9 confirms this fact. Nevertheless, for map *albert-b-laser*, the 2nd group of discretization points minimizes the most the cost, for *utk-claxton* is the 4th group, for *fr079* the cost is minimized in the 2nd execution, for *ubremen-cartesium* is the 3rd one.

The relatively high standard deviation is motivated by the fact that it is dependent on the starting positions of the two players. It can happen that the detection phase ends immediately if the pursuer and the evader have already line-of-sight, resulting in all the values being equal to 0. At the other extreme, the pursuer is forced to clear all the rooms before detecting its adversary because they start from opposite sides of the map. In between, there are all the possible cases. This is a phenomenon that occurs also when the capturing phase is considered, as shown in Section 6.3 and Section 6.4.

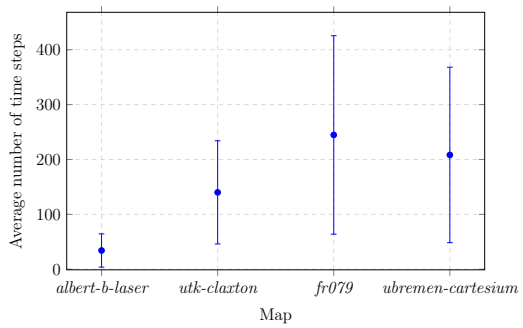
Another important value to consider is the computational time necessary to complete the execution. Figure 6.7 reports the average time and standard deviation for each map and execution. Also in this case, the variable doesn't depend much on the number of discretization points.

For the map *albert-b-laser* the 1st group results in the best value, for *utk-claxton* the 2nd one, while for *fr079* the 1st group. Finally, regarding the map *ubremen-cartesium* the 1st group minimizes the computational time.

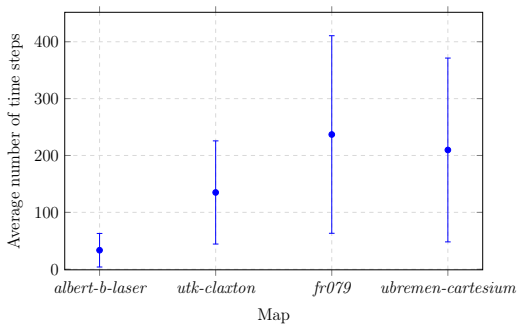
Also in this case, a high standard deviation is observed, for the same motivations explained before.

Finally, Figure 6.8 reports the number of times where at least in one time step the grid decomposition is used instead of the visibility-based one when the indistinguishability issue applies. Even in this case the number does not seem to depend much on the number of discretization points. Nevertheless the fraction with respect to the total number of runs for each map (see Table 6.5 for totals) is usually quite high with respect to the total number of executions, as Figure 6.9 shows.

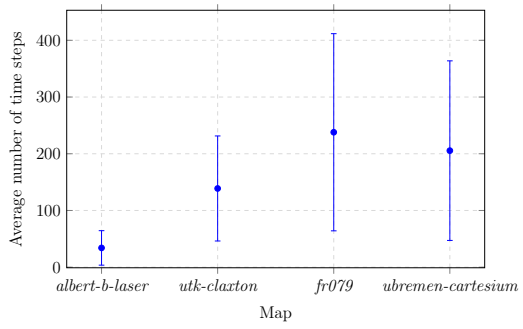
Based on the data acquired, for *albert-b-laser* and *utk-claxton* the 2nd group with 10 and 20 discretization points, respectively, seems to provide the fairest balance among all the figures being evaluated. Regarding *fr079*, 10 discretization points provide the best balance across the evaluated parameters. Considering *ubremen-cartesium*, 5 discretization points seem to guarantee a good balance across the evaluated parameters. These values are selected for the execution of the complete algorithm, which is analyzed in the next sections.



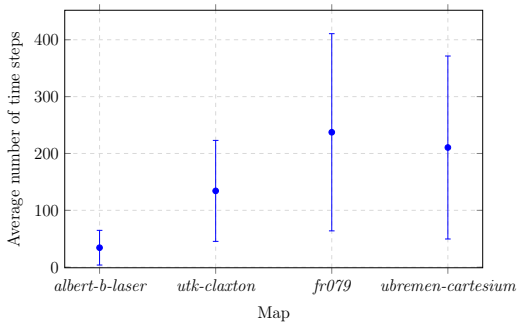
(a) 1st group



(b) 2nd group

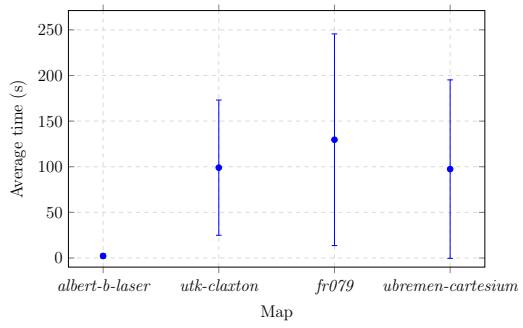


(c) 3rd group

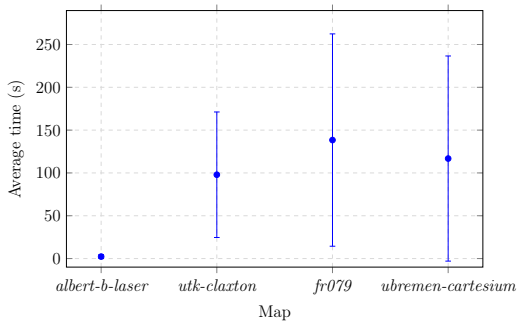


(d) 4th group

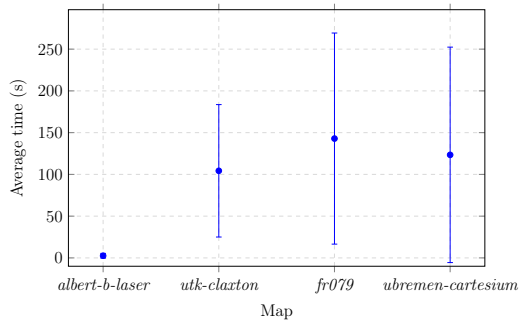
Figure 6.6: Average and standard deviation of time steps needed for detecting the evader changing discretization points (see Table 6.2).



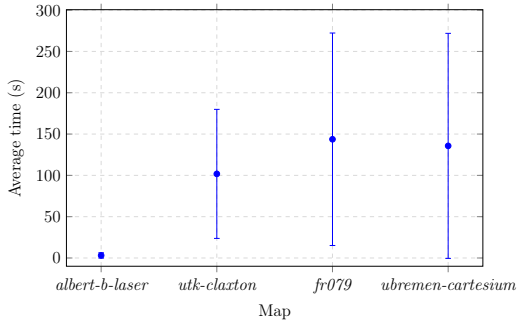
(a) 1st group



(b) 2nd group

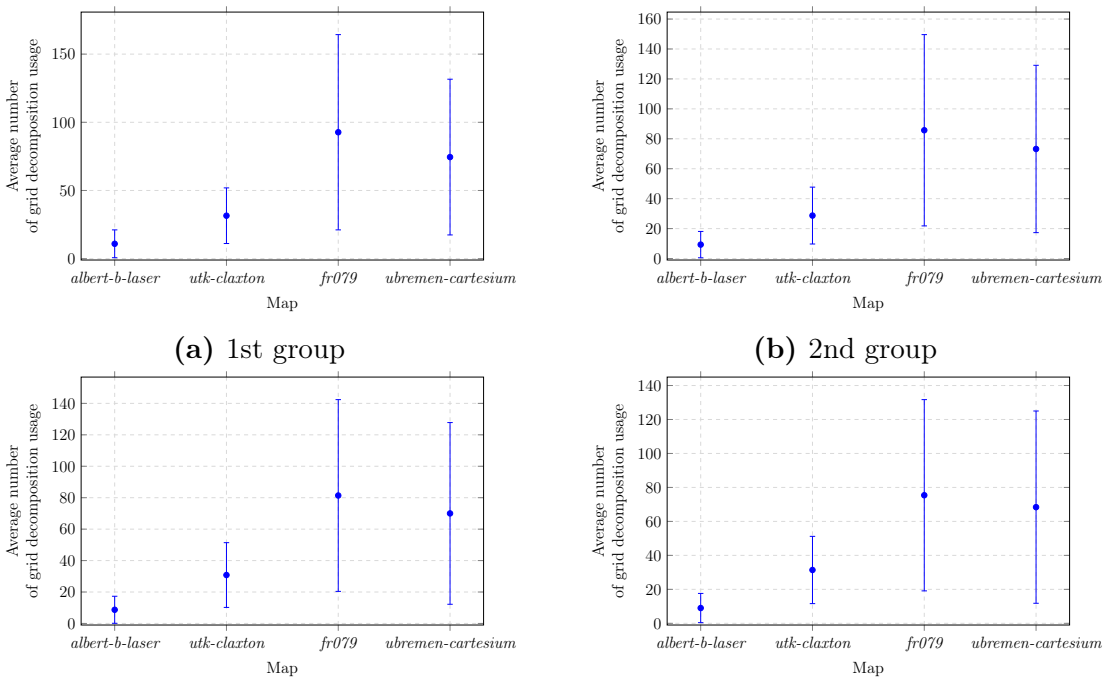


(c) 3rd group



(d) 4th group

Figure 6.7: Average and standard deviation of computational time for detection algorithm with different discretization points (see Table 6.2).



(a) 1st group **(b)** 2nd group

(c) 3rd group **(d)** 4th group

Figure 6.8: Average and standard deviation of number of times grid decomposition is used in the detection algorithm as a function of discretization points (see Table 6.2).

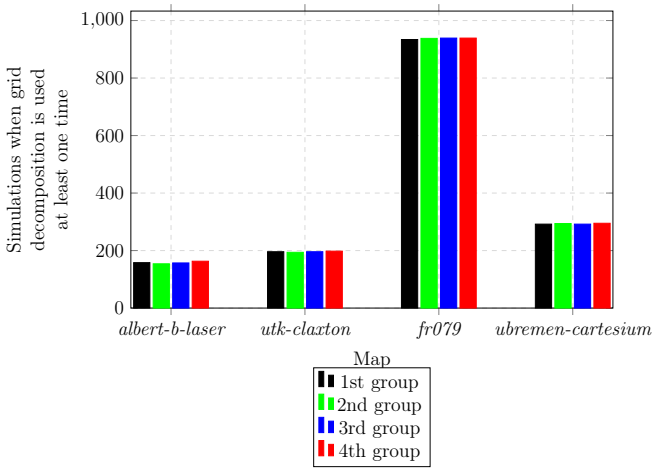


Figure 6.9: Number of simulations where at least in one time step the grid decomposition is used replacing the visibility-based one.

6.3 Minimax with different speeds

Following the discussion about the number of discretization points regarding the detection phase, for each environment, the preferred value is selected and then utilized for each subsequent batch experiment. For every map the number of points selected guarantee a fair discretization of the environments using the visibility-based decomposition.

This section compares the performance of Minimax as a function of the speed of the robots. The first set of experiments is performed using velocity $\nu = 0.10\,\mathrm{m/s}$, which is the same for all the maps, whereas the second one has higher values as a function of the environment. These velocities are reported in Table 6.4. In both scenarios all the settings completed successfully within the allocated timeout, empirically proving $\mathcal{H}_3$ and $\mathcal{H}_4$ (see Section 3.2). Some evaluation is conducted removing the speedup techniques of Section 4.3.2, but most of the experiments fail to terminate before the expiration of the timeout.

Figure 6.11 **(a)**, which plots the left hand side of Table 6.6, shows the average number of time steps, together with its standard deviation. In general, the values are lower when $\nu > 0.10\,\mathrm{m/s}$. This is mainly due to the fact that the game is more fast paced, resulting in goal nodes being reached in less time steps, and thus having lower depth in the search tree that impacts directly the performance of the Minimax algorithm.

Moreover the pursuer is capable to apply the speed ups techniques more effectively. In particular, this holds for the novel pruning technique introduced in this thesis because the pursuer is able to discard branches even earlier than in the case with $\nu = 0.10\,\mathrm{m/s}$.

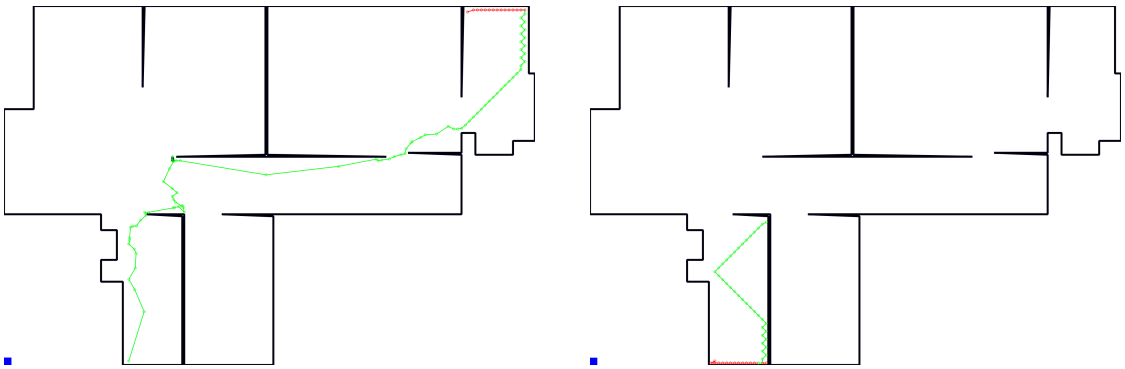
This phenomenon is also observable in the computational time (Figure 6.11 **(b)**), that plots the right hand side of Table 6.6, where in every map, the version with speed $\nu > 0.10\,\mathrm{m/s}$ has lower values.

Considering the case with $\nu = 0.10\,\mathrm{m/s}$, the performance of the methods in the environments *albert-b-laser* and *utk-claxton* is compared with data provided in [69], showing better results in terms of both average and standard deviation regarding cost and computational time in both maps thanks to the novel pruning technique introduced in this thesis.

Even in this case, the previous observations about the behavior of the standard deviation applies. Figures 6.10 **(a)** and **(b)** show two extreme instances computed on *albert-b-laser*. The former involving a high computational time while the latter a significantly lower one. A number of instances fall in either one of these two categories, contributing to the relatively high standard deviation. Of course, there are also settings that fall in the mixed case having cost and/or computational time aligned with the average reported in Figur 6.10. Notice in Figure 6.10 **(b)** the "zigzag" behavior of the pursuer: while closing the distance with the evader, it moves towards the center of room in order to guard a potential escape path. Then, after the evader realizes this, it moves towards the vertex on its right but the pursues follows precisely these movements while it continuously reduces the distance with its adversary. Eventually, it is able to trap the evader and then capture it.

Table 6.6: Results with average and standard deviation over the samples for each map in both configurations of the movement speed of the robots.

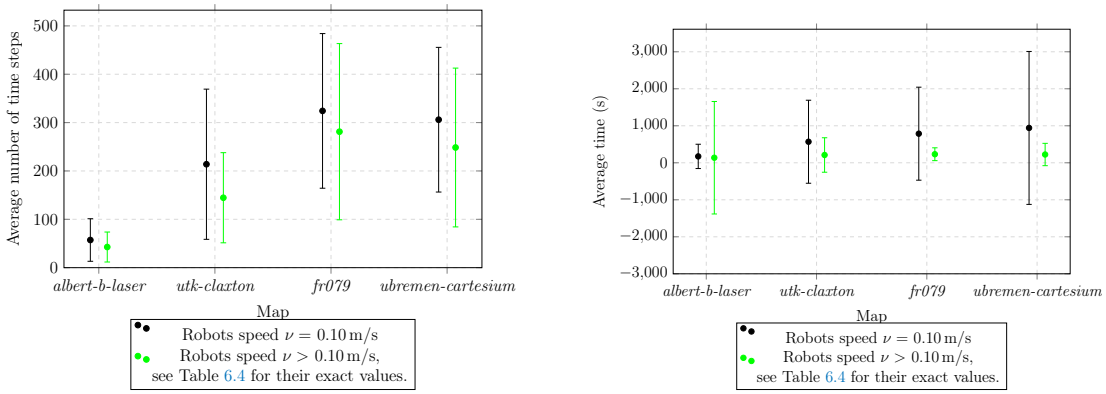
	$\nu = 0.10\,\mathrm{m/s}$		$\nu > 0.10\,\mathrm{m/s}$	
Map	Number of time steps	Computation time (s)	Number of time steps	Computation time (s)
albert-b-laser	57.2 (44.0)	172.8 (328.4)	42.7 (31.0)	136.7 (1520.0)
utk-claxton	213.9 (155.3)	568.8 (1121.7)	144.6 (93.2)	210.7 (465.7)
fr079	324.2 (159.9)	786.4 (1256.7)	281.2 (182.2)	231.3 (173.4)
ubremen-cartesium	306 (149.6)	942 (2066.4)	248.5 (164.2)	223.7 (301.7)



(a) Example setting with high number of time steps.

(b) Another setting, this time with a low number of time steps.

Figure 6.10: Comparing two opposite settings executed in the map *albert-b-laser*. The pursuer walks along the green path, while the evader's route is the red one.



(a) Comparing the average number of time steps with speed $\nu = 0.10\,\mathrm{m/s}$ and $\nu > 0.10\,\mathrm{m/s}$.

(b) Comparing the average computational time with speed $\nu = 0.10\,\mathrm{m/s}$ and $\nu > 0.10\,\mathrm{m/s}$.

Figure 6.11: Plots of the values reported in Table 6.6.

6.4 Comparing Monte-Carlo Tree Search with Minimax

The last group of experiments involves the comparison of the performance on each map between Minimax and two different configuration of MCTS: the first one with low maximum simulation depth and timeout, whereas the second one involves a higher maximum simulation depth and timeout. The executions are further divided into two scenarios: the first one using Monte-Carlo Tree Search for computing the strategies of both players, and a second group, where MCTS is used only for the pursuer whereas the evader executes Minimax.

6.4.1 First scenario: both players use Monte-Carlo Tree Search

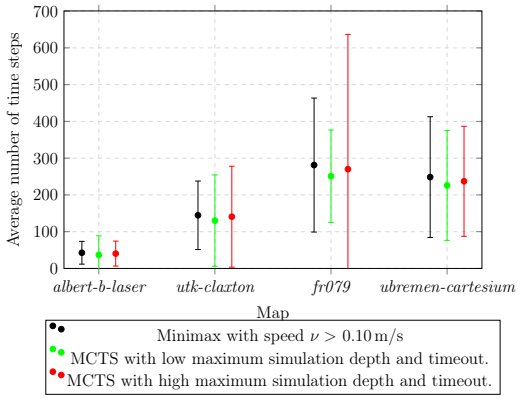
Table 6.7 summarizes the results and Figure 6.12 plots them using the notation already presented in previous plots. As the reader can notice, both versions of MCTS have in general higher computational time with respect to Minimax (right hand side of Table 6.6). This is not easily justifiable, but one possible reason is that a single random simulation requires a fairly relevant amount of time and, therefore, the allocated timeout for a single iteration of the algorithm expires often. Profiling tools highlights that the heuristic function is the bottleneck that causes some slow downs. This happens in general for every the algorithm that gets executed but seems to affect the most MCTS because of the general low amount of time in which the algorithm has to make a decision.

In the first set of experiments the average number of time steps is lower than the Minimax one because, in MCTS, the "goodness" of the moves selected, highly depends on the number of simulations. Unfortunately, the problem highlighted before, implies that both players play sub-optimally and the capture condition is established in fewer moves.

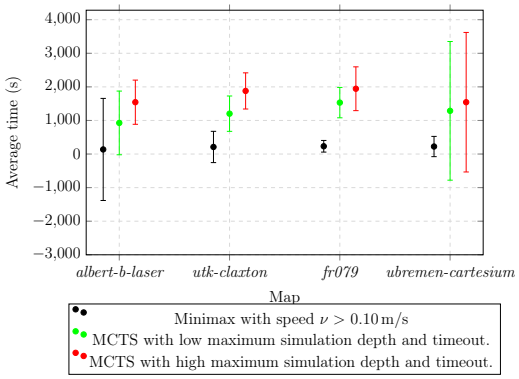
The situation improves when both maximum simulation depth and timeout are increased because, this time, MCTS is able to simulate more playouts and, among them, more reach a goal node, affecting the decision of the selection of the next move more effectively. The direct consequence is that the players play strategies that are more similar to the optimal ones computed with Minimax. Therefore, the results show that the average cost is close to the Minimax case but at expense of increasing the computational time. Nevertheless, the MCTS algorithm manages to complete all the settings within the allocated timeout showing a good amount of robustness against the environments selected for the experiments.

Table 6.7: Results with average and standard deviation over the samples for each map in both configurations regarding the relevant parameters in MCTS.

	Low simulation depth and timeout		High simulation depth and timeout	
Map	Number of time steps	Computation time (s)	Number of time steps	Computation time (s)
albert-b-laser	37.0 (51.7)	926.4 (948.4)	40.4 (34.0)	1543.7 (658.0)
utk-claxton	130.2 (124.3)	1202.1 (526.4)	140.8 (137.3)	1879.2 (538.4)
fr079	250.9 (125.7)	1531.2 (452.0)	270.1 (366.7)	1944.3 (652.2)
ubremen-cartesium	225.9 (149.5)	1286.3 (2064.8)	237.1 (149.6)	1543.4 (2077.5)



(a) Comparison on the average number of time steps.



(b) Comparing the average computational time.

Figure 6.12: Plots of the values reported in Table 6.7 when both players compute their strategy using MCTS.

6.4.2 Second scenario: pursuer uses Monte-Carlo Tree Search and evader uses Minimax

As expected, the computational time in this mixed scenario is higher with respect to the previous scenario. This is reasonable because, while the pursuer is playing sub-optimally in general, the evader executes Minimax at every turn. Moreover, a single execution of Minimax on average lasts more than the timeout allocated for a single turn of MCTS resulting in an increase of the total execution time as well.

Table 6.8 wraps up the results in a similar manner as in the first scenario, and Figure 6.13 shows a comparison between such values and the benchmark pure Minimax case with $\nu > 0.10\,\mathrm{m/s}$.

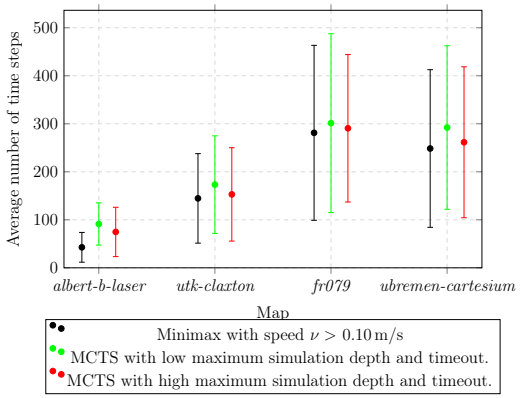
In the first configuration of MCTS, having low simulation depth and timeout, the pursuer exhibits a strategy that consists of sub-optimal moves that result in a slow minimization of the distance between players. This has a direct consequence on augmenting the number of time steps.

The situation improves when the simulation depth and the timeout are increased because, as in the first scenario, MCTS is able to make stronger decisions, selecting moves that close the distance between the players faster. The fact that, in this situation, the number of time steps approaches the values reported in the pure Minimax case, confirms the expected behavior.

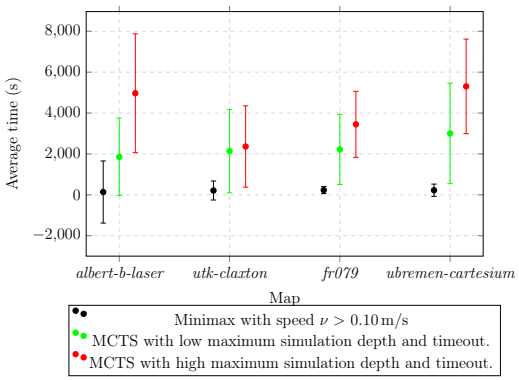
The main downside of this mixed approach is that the increase of computational time can't be neglected, in particular, when the simulation depth and timeout are high, the average time necessary to compute a solution surpasses 1 hour in *albert-b-laser* and *ubremen-cartesium*, but still, it is safely below the global timeout allocated for all the experiments.

Table 6.8: Results with average and standard deviation over the samples for each map in both configurations regarding the relevant parameters in MCTS.

	Low simulation depth and timeout		High simulation depth and timeout	
Map	Number of time steps	Computation time (s)	Number of time steps	Computation time (s)
albert-b-laser	91.3 (44.0)	1858.7 (1893.1)	74.7 (51.4)	4969.0 (2907.5)
utk-claxton	173.3 (101.6)	2135.5 (2036.5)	152.9 (97.2)	2363.9 (1988.8)
fr079	301.3 (186.3)	2221.9 (1715.8)	290.6 (153.6)	3447.7 (1615.6)
ubremen-cartesium	292.2 (170.3)	3004.6 (2457.6)	261.5 (157.2)	5305.1 (2314.4)



(a) Comparison on the average number of time steps.



(b) Comparing the average computational time.

Figure 6.13: Plots of the values reported in Table 6.8, when the pursuer employs MCTS while the evader uses Minimax.

Chapter 7

Conclusions and future work

This thesis has investigated an approach to find optimal solutions to the pursuit-evasion problem with limited visibility in simply-connected polygonal environments. Inspired from [69], this thesis formulates the game as a search problem using Minimax. Moreover, several new insights about the problem are presented. Perhaps the most notable one is a novel technique that compresses the state-space representation resulting in a significant speedup, but also the assumed strategy of the evader, the so called rash evader strategy. In contrast with the state of art presented in Chapter 2, given an environment, the proposed approach computes an optimal strategy for the pursuer that guarantees the capturing of its adversary. After a formal definition of the problem, a more detailed explanation of the context is given, showing that it is reasonable to model indoor environments as simply-connected polygons and also that the rash strategy doesn't penalize the possibilities for the evader. The proposed algorithm solves the problem in two distinct phases. In the first one, called detection, the pursuer clears the environment. The key concepts are two: the so called cleared region, that keeps track of all previous perceptions made by the pursuer, and the cells generated using the visibility-based decomposition, in particular the borders of the cells that are partially visible, that are used by the pursuer to derive the points that could be reached to clear a particular contaminated region. The sequence of moves is derived using a greedy strategy that aims to minimize as fast as possible the distance within the pursuer and the selected point. After the pursuer is able to detect its adversary, the second phase starts. This part uses a classical adversarial algorithm called Minimax, enhanced with several speed ups; some of them already defined in [69], but a novel technique, that allows pruning a branch of the search tree with notable anticipation with respect to the case in which this technique is not applied, provides a dramatic boost in performance. Regarding the research hypotheses, focusing on the optimality of the whole proposed algorithm ($\mathcal{H}_1$), and its two sub-parts, one for each phase ($\mathcal{H}_2$), formulated in Chapter 3, the optimality of Minimax is formally proved in Chapter 4. The estimation utilized in the novel pruning technique is consistent and never overlooks for branches that could be valuable. Unfortunately, it is not easy to derive a similar proof on the first phase of the algorithm remaining an open question. Nevertheless, it is easy to show that the detection algorithm computes a strategy that minimizes the number of times that the pursuer visits a certain point. Being a DFS algorithm, an analysis about the competitive ratio is given, showing

that it is certainly lower than 2, as [23, 41] also demonstrate. This results in $\mathcal{H}_1$ and $\mathcal{H}_2$ being partially answered.

A simulator software built from scratch is used to make the experimental evaluation. These experiments are performed in several indoor environments taken from the Radish repository [34]. The evaluation is composed of four distinct groups of experiments showing that:

- The pursuer is able to detect the evader in every setting regardless of the complexity of the environment and the position of its adversary. The algorithm is not susceptible on the number of discretization points because, in general, they could be indistinguishable (in terms of utility to find the evader) even in the case when the amount of discretization points is low.

- The speed ups and enhancements applied to Minimax contributed dramatically into finding an optimal solution without resorting to more aggressive techniques that sacrifice optimality in favor of speed. When the velocity of the robots is higher, the solution is found even in a lower amount of time.

- Compared to Minimax, when both players use Monte-Carlo Tree Search to compute their strategy, they tend to play sub-optimally, therefore the average number of time steps of the computed solution is lower than in the Minimax case. This is due to the fact that the players have limited amount of time for making a decision based on playouts that don't guarantee to generate an outcome. The situation improves a bit if the timeout and the maximum allowed simulation depth are increased, but the number of steps is still lower than that of Minimax because a good amount of decisions about the selection of the moves are still sub-optimal.

- If MCTS is used only by the pursuer, the number of time steps is higher than in the pure Minimax setting. The motivation is due to the fact that only the pursuer makes sub-optimal moves, therefore it needs more moves in order to catch the evader which conversely is playing optimally using Minimax. The computation time is even higher with respect to the pure MCTS case because, at every turn, the evader has to execute Minimax which, in general, lasts more than a single turn of MCTS.

Overall, the experimental evaluation shows that the performance of this enhanced version of Minimax is superior to that of MCTS, and also with respect to the version introduced in [69] applied on the same environments. This demonstrates that the proposed algorithm is a good technique to solve optimally the pursuit-evasion problem with limited visibility in a simply-connected polygon. All the instances terminated within the allocated timeout and, therefore, the hypotheses about the robustness of the proposed algorithm with respect to the amount of resources used for the computation ($\mathcal{H}_3$) and the environments selected for the experimental evaluation ($\mathcal{H}_4$) are empirically proved.

In order to further validate the methods, additional work is needed. In particular the focuses are:

- **Extend the approach to the general evader case**: in this work only the rash strategy is analyzed, and this assumption is exploited heavily in the detection phase; whenever a region is cleared, the pursuer is assured that the region will be no longer recontaminated unless the evader has been detected at least one time. It could be interesting to remove the assumption and consequently make the detection algorithm more general.

- **Model environment with holes**: sometimes using simple polygons is too restricting. It could be interesting to find a solution of the problem in environments with obstacles when players have different velocities.

- **More realistic simulations**: several strong assumptions are made in order to simplify the simulation, such as the usage of perfect sensors, the ability of the robots to move at full speed every time, batteries that have infinite capacity, and so on. Future work may challenge one or more of these assumptions and study the feasibility and the complexity of adapting the proposed approach to these new scenarios.

- **Extend to more players**: this work has analyzed the problem of pursuit-evasion with a single pursuer and evader. A straightforward question that could be raised is, what happens if we increase the number of players? This could lead to introduce more pursuers that may coordinate in order to clear the environment using perhaps a distributed version of the detection algorithm and then proceed to capture the evader when spotted. This could significantly reduce the number of time steps required to capture the evader, which could be a critical factor, for instance in a search and rescue context.

Appendix A

System documentation

The aim of this appendix is to explain how the algorithms are actually implemented by providing an exhaustive documentation of how the proposed technique works under the hood, what components are allowed to interact with each other, and in which way they are able to do so. After reading this chapter, the reader should be aware about the approach that is utilized for implementing the software, in particular around the concept of maximizing modularity and reusability, which enables the portability of the code to more realistic scenarios.

A.1 Architectural structure

When it comes to documenting software, usually some diagrams are presented through the texts. For a better exchange and understanding of these documents, they should follow a standard structure and some requirements, in particular the diagrams should comply with the set of rules and conventions of the Unified Modeling Language (UML), which, as the name suggests, is a language that helps to model different concepts and behaviors that are typical of software engineering in the industry.

The class diagram, is one the types of diagrams that are part of the original standard and it is probably the most important type of scheme, because, as UML is built using an object paradigm and modern software written using object oriented languages, it naturally maps to the actual code that implements such software. As explained in Section 5.2, the simulation program is implemented in C++ which is an object oriented programming language and therefore the usage of a class diagram to show a complete picture of the most relevant modules of the software is reasonable.

Figure A.1 shows the general architecture of the system; each yellow rectangle is an object that describes and implement a particular entity that it is relevant for the problem that this software tries to solve.

A typical approach to use when implementing complex software architectures is resorting to some kind of patterns. In this case the Model-View-Controller (MVC) it is utilized. It emphasizes a strict separation between three fundamentals parts of an application: the Model which is the data that the software has to manipulate in some way in order to function properly, the View which is the set of interfaces that

are showed to the user in order to enable correct interaction with the software and the Controller which is component where the majority of the business logic of the application resides in.

The typical work flow with this architectural pattern is the following: the View receives the user's input and performs some sanitization or preprocessing if needed, then it sends the proper procedures calls to the Controller which handles the requests, in general it has to resort to the Model in order to retrieve the necessary data to complete such requests but for simple tasks this might not be needed. Then, after the elaboration is completed, the results are sent back to the View and then displayed to the user.

Another scenario involves the triggering of some types of asynchronous conditions in the Model of the application, like an expiration of a timer or a completion of a particular delayed task. In this case the Model notifies the Controller or directly the View whether the change of data requires some kind of computation or not.

By using this paradigm, it is possible to identify the view as the classes called `BatchExperimentDriver` and `MainWindow`, they collect the input from the user from the command line and the Graphical User Interface (GUI) respectively.

These classes interact directly with the `Search`, which corresponds to the Controller and contains all the main algorithms that Chapter 4 shows. These procedures heavily use the set of complex data structures that compose the Model of the application. In the diagram, these classes are `Map`, `GraphNode`, `GraphNode` and `GraphFrontier` for phase 1, `MinMaxState`, `MinMaxNode` and `MinMaxFrontier` for the Minimax algorithm, and finally `MonteCarloState` and `MonteCarloNode` for the Monte Carlo Tree Search.

The main benefits of this approach are the superior modularity of the components with a clear separation of the tasks, along with a higher degree of reusability with respect to other architectural patterns: two different user interfaces are implemented with distinct classes but they interact with the same components so there is no need to implement the same business logic twice. This allows a high degree of reutilization.

A.2 Algorithms implementation

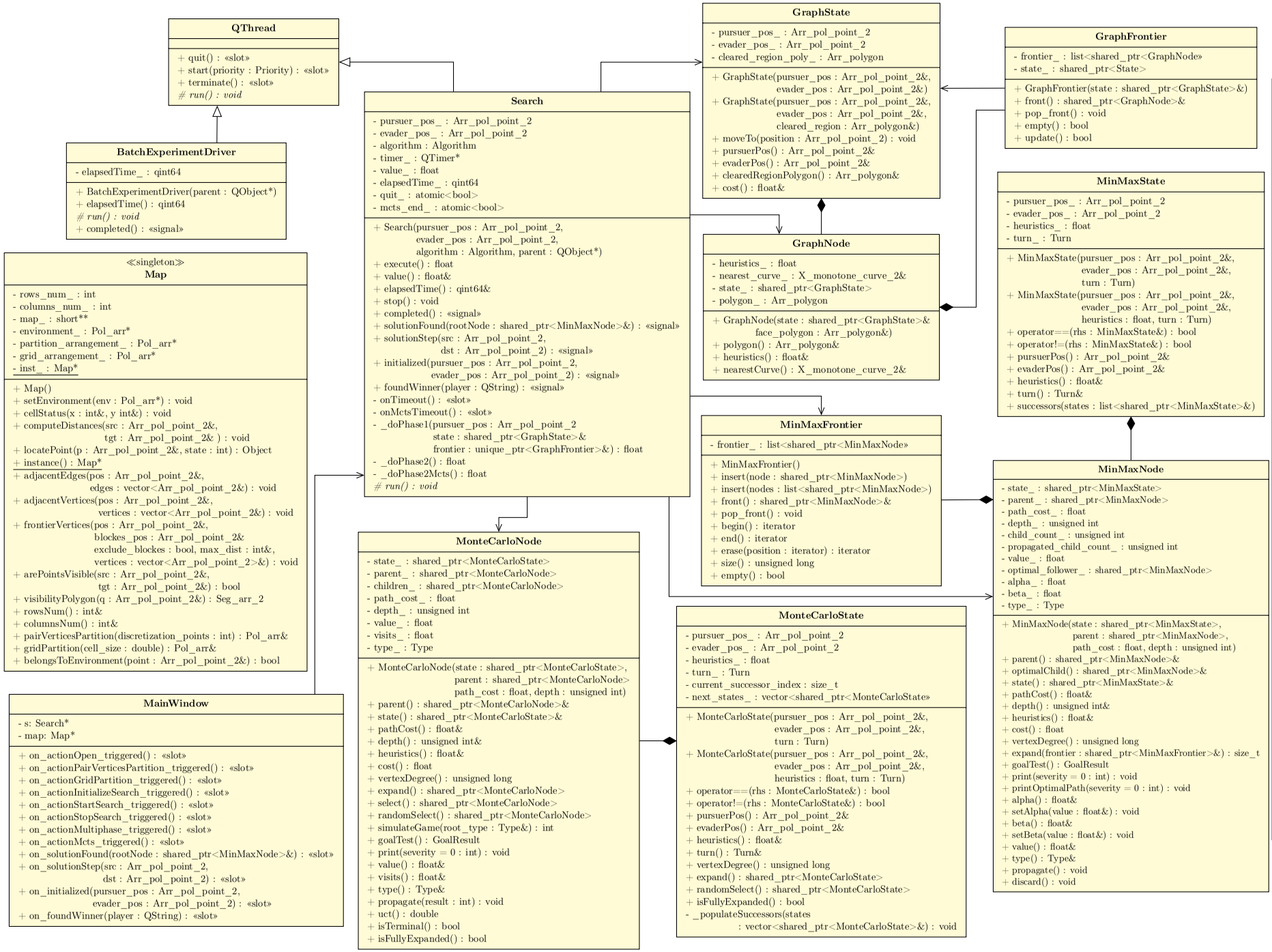
The build system used for compiling the project, is CMake [21], is configured so to generate a separate static library that contains all the executable code regarding the classes that represent Model and Controller. The main benefit of this approach is the possibility to create different executables for each view implemented. In fact, the actual project that manages the application software generates an executable for the GUI mode and another one for the CLI mode.

Additionally, there are some test cases that are run using a different test driver executable that is linked to the same static library as well.

This section explains in more detail the classes with their methods and fields appearing in Figure A.1. In particular, this section focuses on Controller and Model.

Search. This class implements all the algorithms that computes the motion plans both for phase 1 and 2. It inherits from `QThread` since the execution of such

Figure A.1: Class diagram of the software with the main classes and their relationships



procedures should be asynchronous with respect to other threads such as the one that deals with managing the GUI. The relevant methods in this class are (with the parameters being omitted):

- `Search()`, the constructor takes as input the initial position of the players, and the algorithm that it is going to be used,
- `run()`, invoked by Qt when the thread is started with `start()`,
- `execute()`, called in run() surrounded by the timer that measures the execution time of the selected algorithm,
- `value()`, returns the number of total time steps of the solution found,
- `elapsedTime()`, returns the time, in milliseconds, that was needed to compute the selected solution,
- `completed()`, is a signal that gets raised whenever the algorithm is completed and the solution is found,
- `solutionFound()`, signal that gets emitted coupled with `completed()` providing the first node of the solution path ready to be processed by the view,
- `solutionStep()`, during the first phase, and with MCTS every time a player makes a move the algorithm raises this signal to view that in turn displays the movement onto the map or prints it in the standard output,
- `initialized()`, emitted before actually starting the algorithm, it might be useful for the view if it has to setup somethings just before the start of the computation,
- `foundWinner()`, carries the name of the player that has won the game, in the GUI version a dialog message is displayed with the string passed,
- `onTimeout()`, is a slot called when the global timeout expired, it halts immediately the computation and the best candidate solution is provided to the view,
- `onMctsTimeout()`, has a similar behavior to `onTimeout()`, but it aborts the current turn of MCTS and forces the algorithm to select the best promising move depending on the simulations completed during the allocated time,
- `_doPhase1()`, is the method that actually carries the first phase of the whole search algorithm, it can be called during phase 2 with proper parameter setup if the purser loses line-of-sight of the evader and it has to fall back to the detection phase,
- `_doPhase2()`, implements the second phase Minimax algorithm,
- `_doPhase2Mcts()`, contains the code for the second phase Monte Carlo Tree Search algorithm.

Moving to the model, it can be divided in three distinct sub modules, one for each algorithm implemented for either phase 1 or 2.

GraphState. Encodes the current state of the phase 1 algorithm, it holds the positions of the players and manages and update the cleared region for the pursuer, the methods are:

- `GraphState()`, the constructor initializes the state, called before starting the execution of the algorithm, there are two versions: if the detection algorithm is called at the beginning of the search, the version without the already computed cleared region is called, and this information is derived using the visibility region, whereas if phase 1 procedure is invoked when falling back from phase 2, the cleared region is precomputed because, in general it does not coincide with the visibility region,
- `moveTo()`, implements the move of the pursuer to the position based on the selection made by the algorithm in the `Search` class,
- `pursuerPos()`, getter that returns the current pursuer's position,
- `evaderPos()`, similar method as the last one but for the position of the evader,
- `clearedRegionPolygon()`, it returns the polygon that describes the cleared region of the pursuer, used mainly to compute the contaminated regions,
- `cost()`, provides the number of time steps already spent in order to reach the current state.

GraphNode. It encodes a single contaminated region that needs to be cleared:

- `GraphNode()`, is the constructor, it takes the reference of the algorithm state and the polygon that describes this contaminated region,
- `polygon()`, returns the polygon of the contaminated regions,
- `heuristics()`, provides the distance with the nearest discretization point of the segments intersecting the edges of the contaminated region polygon inside the map, with the respect of the pursuer current position,
- `nearestCurve()`, returns the segment that contains the nearest discretization point.

GraphFrontier. It manages the set of nodes, created using the pursuer cleared region, and sorts them in ascending order using the `heuristics()` method of `GraphNode`:

- `GraphFrontier()`, constructs the frontier acquiring a reference to the state of the algorithm,
- `front()`, returns the reference of the nearest contaminated node that will be cleared,

- `pop_front()`, removes the front element, this method has been implemented for consistency with the API of the STL containers,
- `empty()`, returns `true` if the frontier is empty,
- `update()`, after a movement of the pursuer, its cleared region gets updated and therefore the contaminated regions need to get updated, this method carries out this task, creating and deleting nodes whenever is necessary.

Regarding the Minimax algorithm, the relevant classes are the following:

MinMaxState. It models a state of the game in the search tree:

- `MinMaxState()`, instantiates a new state with the pursuer and evader positions, with the player's turn,
- `operator==()`, establishes an equality relationship against other instances of `MinMaxState`,
- `operator!=()`, negation of the previous method,
- `pursuerPos()`, returns the pursuer's position considered in this state,
- `evaderPos()`, likewise as the previous method but for the evader,
- `heuristics()`, given the position of the players as input, returns their distance computed with the heuristic function,
- `turn()`, returns the player that is allowed to make a move,
- `successors()`, implements the successor function given the current state.

MinMaxNode. It encapsulates the corresponding states, the nodes are connected together with a parent-children relationship to form the actual tree:

- `MinMaxNode()`, constructs a new node in the Minimax search tree, setting up the relationship with its parent and the state that encapsulates, along with its cost in terms of time step and the depth along with the branch that this node belongs to,
- `parent()`, returns the parent node of this node,
- `optimalChild()`, returns the child that propagated the value that this node currently has,
- `state()`, returns the corresponding state,
- `pathCost()`, the value of the time step at this node,
- `depth()`, the depth of this node inside the corresponding branch in the tree,
- `heuristics()`, its an alias of `state().heuristics()`,

- `cost()`, is the summation of `pathCost()` and `heuristics()`,
- `vertexDegree()`, the number of cells that are adjacent to the pursuer's position, used as fall back method for sorting nodes in the frontier when there's a tie in the `heuristics()` value between two distinct nodes,
- `expand()`, calls the successors function on the owned state, creates a new node for every successive state and insert them accordingly in the frontier,
- `goalTest()`, performs the goal test on the current node and returns the outcome,
- `print()`, prints the current node for debugging and/or informative purposes,
- `printOptimalPath()`, starting from this node, it prints it and then recursively descend along the optimal child printing them as well,
- `alpha()`, retrieves the value of alpha for the α-β pruning algorithm,
- `setAlpha()`, sets the new value of alpha that affects successive node,
- `beta()`, retrieves the value of beta,
- `setBeta()`, sets the new value of beta,
- `value()`, the value of this node, propagated from leaf child node upwards till this one,
- `type()`, either `MIN` or `MAX`,
- `propagate()`, propagates the value of this node to the parent, it happens whenever all the child nodes have either their value correctly evaluated or they have been discarded,
- `discard()`, discards the node if the algorithm determines it is not worth to continue expanding it.

MinMaxFrontier.

- `MinMaxFrontier()`, creates a new empty frontier,
- `insert()`, inserts a single new node or a list of new nodes computed during the expansion phase of an already selected node,
- `front()`, returns the most promising node for the current player ready to be evaluated,
- `pop_front()`, just like in the `GraphFrontier()` class, removes the front node and it maintains consistency with the API of the STL containers,
- `begin()`, returns the iterator that points at the beginning of this list,
- `end()`, returns the iterator that points past the end of this list,

- `erase()`, removes the item pointed by the iterator supplied as input of this method and return a the updated iterator if the client code wishes to continue the iteration on this list,
- `size()`, returns the number of elements in this list,
- `empty()`, returns `true` if the frontier is empty.

The last algorithm, MCTS rely on these classes:

MonteCarloState. It is similar to `MinMaxState` and it encodes a state of the Monte Carlo search tree:

- `MonteCarloState()`, construct a state similarly to `MinMaxState`,
- `operator==()`, equality test with another `MonteCarloState`,
- `operator!=()`, negation of the equality operator,
- `pursuerPos()`, returns the pursuer's position in this state,
- `evaderPos()`, retrieves the evader's position in this state,
- `heuristics()`, computes and returns the distances between the players position using the heuristic function defined on the grid,
- `turn()`, the player that is allowed to make an action,
- `vertexDegree()`, number of cells that are adjacent to the position of the pursuer,
- `expand()`, selects the most promising successor state that hasn't been chosen yet,
- `randomSelect()`, given the set of successive states, generates randomly an index from a geometrical distribution and selects the states using that index,
- `isFullyExpanded()`, return `true` if all the next states are expanded using the `expand()` method,
- `_populateSuccessors()`, implements the successor function and fills the list supplied in input with the new states.

MonteCarloNode. As in the Minimax case, it encapsulates an instance of the class `MonteCarloState`, performs the goal test, the expansion an random simulation:

- `MonteCarloNode()`, is the constructor, takes ownership of an instance of `MonteCarloState`, sets up the relationship with its parent, along with path cost and depth,
- `parent()`, returns the parent node,

- `state()`, retrieves the encapsulated `MonteCarloState`,
- `pathCost()`, the number of time steps made to reach this node,
- `depth()`, the depth along the branch in the search tree,
- `heuristics()`, alias for `state().heuristics()`,
- `cost()`, returns the sum of `pathCost()` and `heuristics()`,
- `vertexDegree()`, another alias, this time for `state().vertexDegree()`,
- `expand()`, expands this node, calling the underlying `state().expand()` and constructing the corresponding node accordingly,
- `select()`, applies the selection criteria to this node, in order to chose the most promising child node, by recursively calling this method, a leaf node is reached, ready to be expanded,
- `randomSelect()`, similar to `expand()` excepts that the state returned is randomly selected, used during the simulation phase,
- `simulateGame()`, implements the simulation phase starting from this node, going in depth to at most a fixed value,
- `goalTest()`, performs the goal test,
- `print()`, dumps the information of this node on the standard output,
- `value()`, aggregates the result of the simulation of the children nodes,
- `visits()`, holds the number of times whether the MCTS algorithm visited this node
- `type()`, either `MIN` or `MAX`,
- `propagate()`, transmits the result received from a child node, or generated from a simulation if it is a leaf node, up to the root,
- `uct()`, performs the computation of the UCT1 expression, used for the selection criteria in `select()`,
- `isTerminal()`, returns `true` if this node is terminal,
- `isFullyExpanded()`, alias for `state().isFullyExpanded()`

Map. At last, the Model deals also with the environment selected for running the simulation, the main implemented methods are:

- `Map()`, constructs a new empty map,
- `setEnvironment()`, takes ownership of the environment loaded from the view,
- `cellStatus()`, return 0 if the cell at `x` and `y` is reachable, 1 otherwise,
- `computeDistances()`, heuristic function based on grid decomposition,
- `locatePoint()`, query function that returns the object that lies underneath the point supplied in input, the type of partition considered, whether grid or visibility is specified as a parameter,
- `adjacentEdges()`, computes the adjacent discretization points in the visibility-based decomposition of the point supplied as a parameter,
- `adjacentVertices()`, retrieves the center of the adjacent cells of the grid decomposition of the point supplied as the input parameter,
- `frontierVertices()`, similar to `adjacentVertices()`, but considering a speed greater than 1,
- `arePointsVisible()`, test for line-of-sight visibility,
- `visibilityPolygon()`, computes the visibility region of the point supplied as input,
- `rowsNum()`, number of rows of the matrix that contains the availability of the cells,
- `columnsNum()`, number of columns of the same matrix,
- `pairVerticesPartition()`, computes the visibility-based partition on the environment creating a dedicated CGAL 2D Arrangement with the number of discretization points supplied as input parameter,
- `gridPartition()`, computes the grid-based partition constructing a new Arrangement with the edge size supplied as input,
- `belongsToEnvironment()`, returns `true`, if the point supplied in input is a vertex or sits on an edge of the boundary of the environment.

For analyzing and debugging purposes, the searches are logged through the standard output using glog [18]. The log messages are also saved to a specific file so they can be opened in at another time for further analysis.

A.3 GUI implementation

Section 5.1.2 introduces the Graphical User Interface and how it works. The C++ class that implements these functionalities is `MainWindow` as shown in Figure A.1. The following list illustrates the main methods that are also present in the class diagram:

- `MainWindow()`, constructs a new window ready to be shown,
- `on_actionOpen_triggered()`, is a slot that it is invoked whenever the user clicks on «File» and then «Open», an open file dialog pops up asking for selecting a map, a log file of a previously completed search,
- `on_actionPairVerticesPartition_triggered()`, invoked when the users selects the matching name action in «Partition» menu, asks for the number of discretization point and then calls `Map.pairVerticesPartition()`,
- `on_actionGridPartition_triggered()`, similar as the previous slot excepts that it is the entry point for constructing the grid decomposition of the environment,
- `on_actionInitializeSearch_triggered()`, called when user clicks on the corresponding action in «Search» menu
- `on_actionStartSearch_triggered()`, starts the search algorithm
- `on_actionStopSearch_triggered()`, stops the search algorithm,
- `on_actionMultiphase_triggered()`, selects Minimax as the second phase algorithm,
- `on_actionMcts_triggered()`, selects Monte Carlo Tree Search as the second phase algorithm,
- `on_solutionFound()`, catches the corresponding signal raised by the `Search` class, fired at the end of the second phase for Minimax. Starting from the first step of the optimal solution computed, is walks down through the optimal children, the entire sequences of actions of both players are drawn in the map widget,
- `on_solutionStep()`, listens for the matching signal in the `Search` class, which is raised every time a player makes a step, which happens in phase 1 and MCTS after an iteration of the inner loop in Algorithm 4,
- `on_initialized()`, handles the signal `Search.initialized()` which is fired just before starting the computation of the algorithm,
- `on_foundWinner()`, displays a dialog, like in Figure 5.2, catches the signal `Search.foundWinner()` which is raised after the algorithm determines which player has won the simulation.

A.4 CLI implementation

Section 5.1.3 presents the module that encapsulates the CLI. Besides the `main` function that sets up the algorithm for a single search, like the one in Figure 5.3, the class `BatchExperimentDriver` enables automatic creation and execution of many searches carrying out all the necessary configuration of the parameters for each instance of the simulation:

- `BatchExperimentDriver()`, constructs a new instance of this class,
- `run()`, reimplemented from `QThread`, loads an experiment file, creates and executes all the simulations, collecting the results and saving them in a dedicated `csv` file,
- `elapsedTime()`, retrieves the total execution time by aggregating all durations of the instances that are included in the experiment file supplied to the class
- `completed()`, emitted when all the simulations are completed in order to inform the `main` to terminate the program freeing all the remaining resources.

Appendix B

Additional tools developed

This section illustrates a couple of Python scripts that helped the creation of the map files with the correct format that is supported for the simulation software, a `.dat` file. These tools help transform a raster image to a textual file that contains the set of control points that describe the environment present in the initial file together with the correct scale.

The first step of the process is implemented in Listing B.1. The Python bindings of the OpenCV2 library [63] derive a matrix representation of the pixels contained in `infile`. In order to disambiguate between the background and the actual path, the image is assumed to have the contour of the map colored with black and a white background. Then an algorithm accesses the cells of this matrix deriving the set of control points. Every time there is a change of direction a point is added and then saved to `outfile`.

Listing B.1: *raster_to_poly.py* script.

```
1  #! /usr/bin/env python
2  import numpy as np
3  import cv2
4  import plac
5
6
7  LEFT = 0
8  TOP = 1
9  RIGHT = 2
10 BOTTOM = 3
11 dirs = ['LEFT', 'TOP', 'RIGHT', 'BOTTOM']
12
13
14 def yflip(points, matrix):
15     flipped = []
16
17     for p in points:
18         flipped.append([matrix.shape[0] - p[0] - 1, p
               [1]])
19
```

```
20      return flipped
21
22
23  def neighbors(pos, matrix):
24      dneigh = [[0, -1], [-1, 0], [0, 1], [1, 0]]
25      l1 = []
26      l2 = []
27      for dn in dneigh:
28          n = [sum(x) for x in zip(pos, dn)]
29          l1.append(n)
30          if n[0] < 0 or n[1] < 0 or n[0] == matrix.shape
                [0] or n[1] == matrix.shape[1]:
31              l2.append(255)
32          else:
33              l2.append(matrix[n[0], n[1]])
34      return list(zip(l1, l2))
35
36
37  def get_directions(start, point, direction, matrix,
        dirs_array, resume):
38      saved_dir = direction
39      print("%s: directions(%s, %s, %s, ...)" % (resume,
            start, point, dirs[direction]))
40      cur = neighbors(point, matrix)[direction][0]
41
42      while cur != start and cur != point:
43          points = neighbors(cur, matrix)
44          if direction == TOP:
45              if points[LEFT][1] == 0 and points[RIGHT][1]
                    == 0:
46                  d = []
47                  if get_directions(start, cur, LEFT,
                        matrix, d, resume + 1):
48                      dirs_array.append(LEFT)
49                      dirs_array.extend(d)
50                      print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
51                      return True
52                  elif get_directions(start, cur, RIGHT,
                        matrix, d, resume + 1):
53                      dirs_array.extend(RIGHT)
54                      dirs_array.extend(d)
55                      print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
56                      return True
57              elif points[LEFT][1] == 0 and points[TOP][1]
                    == 0:
```

```
58                  d = []
59                  if get_directions(start, cur, LEFT,
                        matrix, d, resume + 1):
60                      dirs_array.append(LEFT)
61                      dirs_array.extend(d)
62                      print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
63                      return True
64                  elif get_directions(start, cur, TOP,
                        matrix, d, resume + 1):
65                      dirs_array.append(TOP)
66                      dirs_array.extend(d)
67                      print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
68                      return True
69              elif points[RIGHT][1] == 0 and points[TOP
                    ][1] == 0:
70                  d = []
71                  if get_directions(start, cur, RIGHT,
                        matrix, d, resume + 1):
72                      dirs_array.append(RIGHT)
73                      dirs_array.extend(d)
74                      print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
75                      return True
76                  elif get_directions(start, cur, TOP,
                        matrix, d, resume + 1):
77                      dirs_array.append(TOP)
78                      dirs_array.extend(d)
79                      print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
80                      return True
81              elif points[LEFT][1] == 0:
82                  direction = LEFT
83              elif points[RIGHT][1] == 0:
84                  direction = RIGHT
85              elif points[TOP][1] > 0:
86                  print("%s, %s: -> False" % (resume, dirs
                        [saved_dir]))
87                  return False
88          elif direction == BOTTOM:
89              if points[LEFT][1] == 0 and points[RIGHT][1]
                    == 0:
90                  d = []
91                  if get_directions(start, cur, LEFT,
                        matrix, d, resume + 1):
92                      dirs_array.append(LEFT)
```

```
93                      dirs_array.extend(d)
94                      print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
95                      return True
96                  elif get_directions(start, cur, RIGHT,
                        matrix, d, resume + 1):
97                      dirs_array.append(RIGHT)
98                      dirs_array.extend(d)
99                      print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
100                     return True
101             elif points[LEFT][1] == 0 and points[BOTTOM
                    ][1] == 0:
102                 d = []
103                 if get_directions(start, cur, LEFT,
                        matrix, d, resume + 1):
104                     dirs_array.append(LEFT)
105                     dirs_array.extend(d)
106                     print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
107                     return True
108                 elif get_directions(start, cur, BOTTOM,
                        matrix, d, resume + 1):
109                     dirs_array.append(BOTTOM)
110                     dirs_array.extend(d)
111                     print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
112                     return True
113             elif points[RIGHT][1] == 0 and points[BOTTOM
                    ][1] == 0:
114                 d = []
115                 if get_directions(start, cur, RIGHT,
                        matrix, d, resume + 1):
116                     dirs_array.append(RIGHT)
117                     dirs_array.extend(d)
118                     print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
119                     return True
120                 elif get_directions(start, cur, BOTTOM,
                        matrix, d, resume + 1):
121                     dirs_array.append(BOTTOM)
122                     dirs_array.extend(d)
123                     print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
124                     return True
125             elif points[LEFT][1] == 0:
126                 direction = LEFT
```

```
127             elif points[RIGHT][1] == 0:
128                 direction = RIGHT
129             elif points[BOTTOM][1] > 0:
130                 print("%s, %s: -> False" % (resume, dirs
                        [saved_dir]))
131                 return False
132         elif direction == LEFT:
133             if points[TOP][1] == 0 and points[BOTTOM][1]
                    == 0:
134                 d = []
135                 if get_directions(start, cur, TOP,
                        matrix, d, resume + 1):
136                     dirs_array.append(TOP)
137                     dirs_array.extend(d)
138                     print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
139                     return True
140                 elif get_directions(start, cur, BOTTOM,
                        matrix, d, resume + 1):
141                     dirs_array.append(BOTTOM)
142                     dirs_array.extend(d)
143                     print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
144                     return True
145             elif points[BOTTOM][1] == 0 and points[LEFT
                    ][1] == 0:
146                 d = []
147                 if get_directions(start, cur, BOTTOM,
                        matrix, d, resume + 1):
148                     dirs_array.append(BOTTOM)
149                     dirs_array.extend(d)
150                     print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
151                     return True
152                 elif get_directions(start, cur, LEFT,
                        matrix, d, resume + 1):
153                     dirs_array.append(LEFT)
154                     dirs_array.extend(d)
155                     print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
156                     return True
157             elif points[TOP][1] == 0 and points[LEFT][1]
                    == 0:
158                 d = []
159                 if get_directions(start, cur, TOP,
                        matrix, d, resume + 1):
160                     dirs_array.append(TOP)
```

```
161                     dirs_array.extend(d)
162                     print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
163                     return True
164                 elif get_directions(start, cur, LEFT,
                        matrix, d, resume + 1):
165                     dirs_array.append(LEFT)
166                     dirs_array.extend(d)
167                     print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
168                     return True
169             elif points[TOP][1] == 0:
170                 direction = TOP
171             elif points[BOTTOM][1] == 0:
172                 direction = BOTTOM
173             elif points[LEFT][1] > 0:
174                 print("%s, %s: -> False" % (resume, dirs
                        [saved_dir]))
175                 return False
176         elif direction == RIGHT:
177             if points[TOP][1] == 0 and points[BOTTOM][1]
                    == 0:
178                 d = []
179                 if get_directions(start, cur, TOP,
                        matrix, d, resume + 1):
180                     dirs_array.append(TOP)
181                     dirs_array.extend(d)
182                     print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
183                     return True
184                 elif get_directions(start, cur, BOTTOM,
                        matrix, d, resume + 1):
185                     dirs_array.append(BOTTOM)
186                     dirs_array.extend(d)
187                     print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
188                     return True
189             elif points[BOTTOM][1] == 0 and points[RIGHT
                    ][1] == 0:
190                 d = []
191                 if get_directions(start, cur, BOTTOM,
                        matrix, d, resume + 1):
192                     dirs_array.append(BOTTOM)
193                     dirs_array.extend(d)
194                     print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
195                     return True
```

```
196                 elif get_directions(start, cur, RIGHT,
                        matrix, d, resume + 1):
197                     dirs_array.append(RIGHT)
198                     dirs_array.extend(d)
199                     print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
200                     return True
201             elif points[TOP][1] == 0 and points[RIGHT
                    ][1] == 0:
202                 d = []
203                 if get_directions(start, cur, TOP,
                        matrix, d, resume + 1):
204                     dirs_array.append(TOP)
205                     dirs_array.extend(d)
206                     print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
207                     return True
208                 elif get_directions(start, cur, RIGHT,
                        matrix, d, resume + 1):
209                     dirs_array.append(RIGHT)
210                     dirs_array.extend(d)
211                     print("%s, %s: -> True" % (resume,
                            dirs[saved_dir]))
212                     return True
213             elif points[TOP][1] == 0:
214                 direction = TOP
215             elif points[BOTTOM][1] == 0:
216                 direction = BOTTOM
217             elif points[RIGHT][1] > 0:
218                 print("%s, %s: -> False" % (resume, dirs
                        [saved_dir]))
219                 return False
220         cur = neighbors(cur, matrix)[direction][0]
221
222     print("%s, %s: -> True" % (resume, dirs[saved_dir]))
223     return True
224
225
226 def enumerate_valid_neighbors(neighs):
227     return [x for x in enumerate(neighs) if neighs[x
            [0]][1] == 0]
228
229
230 def find_neighbor(neighs, direction):
231     return [x[1] for x in neighs if x[0] == direction]
232
233
```

```
234 def filter_opposite(neighs, direction):
235     opp = -1
236     if direction == TOP:
237         opp = BOTTOM
238     elif direction == BOTTOM:
239         opp = TOP
240     elif direction == LEFT:
241         opp = RIGHT
242     elif direction == RIGHT:
243         opp = LEFT
244 
245     if opp > -1:
246         return remove_direction(neighs, opp)
247 
248     return neighs
249 
250 
251 def remove_direction(neighs, direction):
252     return [x for x in neighs if x[0] != direction]
253 
254 
255 def walk_to_dead_end(point, direction, matrix):
256     print("walk_to_dead_end(%s, %s, ...)" % (point, dirs
        [direction]))
257     control_points = []
258     while True:
259         points = filter_opposite(
            enumerate_valid_neighbors(neighbors(point,
            matrix)), direction)
260 
261         if len(points) == 0:
262             control_points.append(point)
263             break
264 
265         if len(points) >= 2:
266             control_points.append(point)
267             points = remove_direction(points, direction)
268             for p in points:
269                 control_points.extend(walk_to_dead_end(p
                    [1][0], p[0], matrix))
270                 control_points.append(point)
271         elif len(find_neighbor(points, direction)) == 0:
272             control_points.append(point)
273             direction = points[0][0]
274 
275         point = find_neighbor(enumerate_valid_neighbors(
            neighbors(point, matrix)), direction)
```

```
276
277         if not point:
278             break
279
280         point = point[0][0]
281
282     control_points.extend(list(reversed(control_points))
        [1:])
283
284     return control_points
285
286
287 def walk_along_border(start, direction, matrix,
    directions):
288     cur = find_neighbor(enumerate_valid_neighbors(
        neighbors(start, matrix)), direction)[0][0]
289     control_points = []
290
291     while cur != start:
292         print("Reached point %s, direction %s" % (cur,
            dirs[direction]))
293         points = filter_opposite(
            enumerate_valid_neighbors(neighbors(cur,
            matrix)), direction)
294
295         if len(points) >= 2:
296             control_points.append(cur)
297             direction = directions.pop(0)
298             points = remove_direction(points, direction)
299             for p in points:
300                 control_points.extend(walk_to_dead_end(p
                    [1][0], p[0], matrix))
301                 control_points.append(cur)
302         elif len(find_neighbor(points, direction)) == 0:
303             control_points.append(cur)
304             direction = points[0][0]
305
306         cur = find_neighbor(enumerate_valid_neighbors(
            neighbors(cur, matrix)), direction)[0][0]
307
308     control_points.append(start)
309     return control_points
310
311
312 def resize(points, scale):
313     for p in points:
314         p[0] *= scale
```

```
315         p[1] *= scale
316
317
318 @plac.annotations(
319     cellsize=('Cell size in pixels', 'positional', None,
        float),
320 )
321 def main(infile, outfile, cellsize):
322     source = cv2.imread(infile, 0)
323
324     resized = cv2.resize(source, (0, 0), fx=1 / cellsize
        , fy=1 / cellsize)
325     start = [resized.shape[0] - 1, np.argwhere(resized[
        resized.shape[0] - 1, :] == 0)[0][0]]
326     directions = []
327     get_directions(start, start, TOP, resized,
        directions, 1)
328
329     for d in directions:
330         print("%s" % dirs[d])
331
332
333     control_points = [start]
334     control_points.extend(walk_along_border(start, TOP,
        resized, directions))
335
336     print(control_points)
337
338     control_points = yflip(control_points, resized)
339     resize(control_points, cellsize)
340
341     print(control_points)
342
343     out = open(outfile, 'w')
344
345     out.write("1\n")
346     out.write("%s " % len(control_points))
347
348     for c in control_points:
349         out.write("%s %s " % (str(int(c[1])), str(int(c
            [0]))))
350
351     out.write("\n")
352
353     out.close()
354
355 if __name__ == '__main__':
```

```
356     plac.call(main)
```

The second, and final, step of the process takes as input an already generated map `.dat` file (`infile`), and multiplies the coordinates of the control points with the `scale` parameter. Eventually a new map file is saved to `outfile`. The script code is illustrated in Listing B.2.

Listing B.2: *scale_poly.py* script.

```
1  #! /usr/bin/env python
2
3  import plac
4
5
6  @plac.annotations(
7      scale=('Map scale', 'option', 's', int)
8  )
9  def main(infile, outfile, scale):
10     indat = open(infile, 'r')
11     outdat = open(outfile, 'w')
12
13     outdat.write(indat.readline())
14     points = indat.readline().strip().split(" ")
15
16     outdat.write("%s " % points.pop(0))
17
18     outdat.write("%s\n" % " ".join([str(int(x) * scale)
           for x in points]))
19
20     indat.close()
21     outdat.close()
22
23 if __name__ == '__main__':
24     plac.call(main)
```

Bibliography

[1] M. Adler, H. Räcke, N. Sivadasan, C. Sohler, and B. Vöcking. "Randomized pursuit-evasion in graphs". In: *Combinatorics, Probability and Computing* 12.3 (2003), pp. 225–244 (cit. on pp. 11, 21).

[2] M. Aigner and M. Fromme. "A game of cops and robbers". In: *Discrete Applied Mathematics* 8.1 (1984), pp. 1–12 (cit. on pp. 11, 21).

[3] S. Alexander, R. Bishop, and R. Ghrist. "Capture pursuit games on unbounded domains". In: *L'Enseignement Mathématique(2)* 55.1-2 (2009), pp. 103–125 (cit. on pp. 2, 8, 20).

[4] S. Alexander, R. Bishop, and R. Ghrist. "Pursuit and Evasion in Non-convex Domains of Arbitrary Dimensions." In: *Proceedings of Robotics: Science and Systems*. 2006, pp. 113–120 (cit. on pp. 8, 20).

[5] *Alpha Go | DeepMind*. https://deepmind.com/research/alphago/. Accessed: 2017-09-06 (cit. on p. 18).

[6] B. Ames, A. Beveridge, R. Carlson, C. Djang, V. Isler, S. Ragain, and M. Savage. "A leapfrog strategy for pursuit-evasion in a polygonal environment". In: *International Journal of Computational Geometry & Applications* 25.02 (2015), pp. 77–100 (cit. on pp. 8, 20).

[7] F. Amigoni and N. Basilico. "A game theoretical approach to finding optimal strategies for pursuit evasion in grid environments". In: *Proceedings of International Conference on Robotics and Automation (ICRA)*. IEEE. 2012, pp. 2155–2162 (cit. on pp. 12, 21).

[8] T. Asano. "An efficient algorithm for finding the visibility polygon for a polygonal region with holes". In: *The Institute of Electronics, Information and Communication Engineers (IEICE) Transactions (1976-1990)* 68.9 (1985), pp. 557–559 (cit. on p. 18).

[9] T. Başar and G. Olsder. *Dynamic noncooperative game theory*. SIAM, 1998 (cit. on p. 14).

[10] A. Beveridge and Y. Cai. "Two-dimensional pursuit-evasion in a compact domain with piecewise analytic boundary". In: *arXiv preprint arXiv:1505.00297* (2015) (cit. on pp. 8, 20).

[11] D. Bhadauria and V. Isler. "Capturing an Evader in a Polygonal Environment with Obstacles." In: *Proceedings of International Joint Conference on Artificial Intelligence*. 2011, pp. 2054–2059 (cit. on pp. 7, 8, 20).

[12] D. Bhadauria, K. Klein, V. Isler, and S. Suri. "Capturing an evader in polygonal environments with obstacles: The full visibility case". In: *The International Journal of Robotics Research* 31.10 (2012), pp. 1176–1189 (cit. on pp. 8, 20).

[13] S. Bhattacharya and S. Hutchinson. "On the existence of nash equilibrium for a two player pursuit-evasion game with visibility constraints". In: *Algorithmic Foundation of Robotics VIII*. Springer, 2009, pp. 251–265 (cit. on pp. 10, 20).

[14] *Boost C++ Libraries*. http://www.boost.org/. Accessed: 2017-10-05 (cit. on p. 54).

[15] S. Bopardikar, F. Bullo, and J. Hespanha. "Sensing limitations in the Lion and Man problem". In: *Proceedings of American Control Conference (ACC)*. IEEE. 2007, pp. 5958–5963 (cit. on pp. 3, 10, 20).

[16] J. Britnell and M. Wildon. "Finding a princess in a palace: a pursuit-evasion problem". In: *arXiv preprint arXiv:1204.5490* (2012) (cit. on pp. 12, 21).

[17] F. Bungiu, M. Hemmer, J. Hershberger, K. Huang, and A. Kröller. "Efficient computation of visibility polygons". In: *arXiv preprint arXiv:1403.3905* (2014) (cit. on p. 18).

[18] *C++ implementation of the Google logging module*. https://github.com/google/glog. Accessed: 2017-10-15 (cit. on p. 86).

[19] *CARMEN, then Carnegie Mellon Robot Navigation Toolkit*. http://carmen.sourceforge.net/. Accessed: 2017-10-21 (cit. on p. 50).

[20] G. Chaslot. "Monte-carlo tree search". PhD thesis. Maastricht: Universiteit Maastricht, 2010 (cit. on pp. 17, 18, 45).

[21] *CMake*. https://cmake.org/. Accessed: 2017-10-06 (cit. on p. 78).

[22] R. Coulom. "Efficient selectivity and backup operators in Monte-Carlo tree search". In: *Proceedings of International Conference on Computers and Games*. Springer. 2006, pp. 72–83 (cit. on p. 17).

[23] A. Dessmark and A. Pelc. "Optimal graph exploration without good maps". In: *Theoretical Computer Science* 326.1-3 (2004), pp. 343–362 (cit. on pp. 37, 74).

[24] J. Durham, A. Franchi, and F. Bullo. "Distributed pursuit-evasion without mapping or global localization via local frontiers". In: *Autonomous Robots* 32.1 (2012), pp. 81–95 (cit. on pp. 10, 20).

[25] J. Eaton and L. Zadeh. "Optimal Pursuit Strategies in Discrete-State Probabilistic Systems". In: *Journal of Basic Engineering* 84.1 (1962), pp. 23–29 (cit. on p. 12).

[26] B. Gerkey, S. Thrun, and G. Gordon. "Visibility-based pursuit-evasion with limited field of view". In: *The International Journal of Robotics Research* 25.4 (2006), pp. 299–315 (cit. on pp. 10, 18, 20).

[27] A. Goldstein and E. Reingold. "The complexity of pursuit on a graph". In: *Theoretical computer science* 143.1 (1995), pp. 93–112 (cit. on pp. 11, 21).

[28] L. Guibas, J.-C. Latombe, S. LaValle, D. Lin, and R. Motwani. "A visibility-based pursuit-evasion problem". In: *International Journal of Computational Geometry & Applications* 9.04n05 (1999), pp. 471–493 (cit. on pp. 2, 7, 9, 10, 20).

[29] J. Hespanha, M. Prandini, and S. Sastry. "Probabilistic pursuit-evasion games: A one-step Nash approach". In: *Proceedings of the 39th Conference on Decision and Control*. Vol. 3. IEEE. 2000, pp. 2272–2277.

[30] G. Hollinger, A. Kehagias, and S. Singh. "GSST: Anytime guaranteed search". In: *Autonomous Robots* 29.1 (2010), pp. 99–118 (cit. on pp. 12, 21).

[31] G. Hollinger, S. Singh, J. Djugash, and A. Kehagias. "Efficient multi-robot search for a moving target". In: *The International Journal of Robotics Research* 28.2 (2009), pp. 201–219 (cit. on pp. 13, 21).

[32] G. Hollinger, S. Singh, and A. Kehagias. "Improving the efficiency of clearing with multi-agent teams". In: *The International Journal of Robotics Research* 29.8 (2010), pp. 1088–1105 (cit. on pp. 12, 21).

[33] *How To Use Gflags (formerly Google Commandline Flags)*. https://gflags.github.io/gflags/. Accessed: 2017-10-15 (cit. on p. 50).

[34] A. Howard and N. Roy. *The Robotics Data Set Repository (Radish)*. 2003. URL: http://radish.sourceforge.net/ (cit. on pp. 50, 57, 74).

[35] R. Isaacs. *Differential Games*. Wiley, 1965 (cit. on p. 13).

[36] V. Isler, G. Hollinger, and T. Chung. "Search and Pursuit-Evasion in Mobile Robotics, A survey". In: *Autonomous Robots* 31.4 (2011), pp. 299–316 (cit. on p. 5).

[37] V. Isler, S. Kannan, and S. Khanna. "Randomized pursuit-evasion in a polygonal environment". In: *IEEE Transactions on Robotics* 21.5 (2005), pp. 875–884 (cit. on pp. 10, 20).

[38] V. Isler, S. Kannan, and S. Khanna. "Randomized pursuit-evasion with local visibility". In: *SIAM Journal on Discrete Mathematics* 20.1 (2006), pp. 26–41 (cit. on pp. 11, 21).

[39] V. Isler and N. Karnad. "The role of information in the cop-robber game". In: *Theoretical Computer Science* 399.3 (2008), pp. 179–190 (cit. on pp. 11, 21).

[40] B. Joe and R. Simpson. "Corrections to Lee's visibility polygon algorithm". In: *BIT Numerical Mathematics* 27.4 (1987), pp. 458–473 (cit. on p. 18).

[41] T. Kamphans. "Models and algorithms for online exploration and search". PhD thesis. 2011 (cit. on pp. 37, 74).

[42] N. Karnad and V. Isler. "Bearing-only pursuit". In: *Proceedings of International Conference on Robotics and Automation (ICRA)*. IEEE. 2008, pp. 2665–2670 (cit. on pp. 3, 10, 20).

[43] M. Katsev, A. Yershova, B. Tovar, R. Ghrist, and S. LaValle. "Mapping and pursuit-evasion strategies for a simple wall-following robot". In: *IEEE Transactions on Robotics* 27.1 (2011), pp. 113–128 (cit. on pp. 2, 10, 20).

[44] K. Klein and S. Suri. "Capture bounds for visibility-based pursuit evasion". In: *Computational Geometry* 48.3 (2015), pp. 205–220 (cit. on p. 14).

[45] K. Klein and S. Suri. "Pursuit evasion on polyhedral surfaces". In: *Algorithmica* 73.4 (2015), pp. 730–747 (cit. on pp. 8, 20).

[46] L. Kocsis and C. Szepesvári. "Bandit Based Monte-Carlo Planning". In: *Proceedings of European Conference on Machine Learning (ECML)*. 2006, pp. 282–293 (cit. on p. 45).

[47] A. Kolling and S. Carpin. "Pursuit-evasion on trees by robot teams". In: *IEEE Transactions on Robotics* 26.1 (2010), pp. 32–47 (cit. on pp. 11, 21, 37).

[48] A. Kolling and S. Carpin. "The graph-clear problem: definition, theoretical properties and its connections to multirobot aided surveillance". In: *Proceedings of International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2007, pp. 1003–1008 (cit. on pp. 11, 21, 37).

[49] Georgios Konstantinidis and Ath Kehagias. "Selfish Cops and Adversarial Robber: Multi-Player Pursuit Evasion on Graphs". In: *arXiv preprint arXiv:1703.07695* (2017) (cit. on pp. 12, 21).

[50] S. Kopparty and C. Ravishankar. "A framework for pursuit evasion games in Rn". In: *Information Processing Letters* 96.3 (2005), pp. 114–122 (cit. on pp. 2, 8, 20).

[51] H. Lau, S. Huang, and G. Dissanayake. "Optimal search for multiple targets in a built environment". In: *Proceedings of International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2005, pp. 3740–3745 (cit. on pp. 13, 21).

[52] H. Lau, S. Huang, and G. Dissanayake. "Probabilistic search for a moving target in an indoor environment". In: *Proceedings of International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2006, pp. 3393–3398 (cit. on pp. 13, 21).

[53] S. LaValle, D. Lin, L. Guibas, J.-C. Latombe, and R. Motwani. "Finding an unpredictable target in a workspace with obstacles". In: *Proceedings of International Conference on Robotics and Automation (ICRA)*. Vol. 1. IEEE. 1997, pp. 737–742 (cit. on pp. 7, 10, 20).

[54] S. Lazebnik. "Visibility-based pursuit-evasion in three-dimensional environments". In: *University of Illinois at Urbana-Champaign, Tech. Rep.* (2001) (cit. on pp. 10, 20).

[55] D.-T. Lee. "Visibility of a simple polygon". In: *Computer Vision, Graphics, and Image Processing* 22.2 (1983), pp. 207–221 (cit. on p. 18).

[56] V. Lisý, B. Bošanský, and M. Pěchouček. "Anytime algorithms for multiagent visibility-based pursuit-evasion games". In: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 3*. International Foundation for Autonomous Agents and Multiagent Systems. 2012, pp. 1301–1302 (cit. on pp. 17, 18).

[57] J. Littlewood. *A Mathematician's Miscellany, vol.* 1953, pp. 135–136 (cit. on pp. 2, 7, 20).

[58] A. Merz. *The homicidal chauffeur-a differential game*. Tech. rep. SUDAAR-418. Stanford University Department Of Aeronautics and Astronautics, 1971 (cit. on p. 14).

[59] R. Murrieta-Cid, R. Monroy, S. Hutchinson, and J.-P. Laumond. "A complexity result for the pursuit-evasion game of maintaining visibility of a moving evader". In: *Proceedings of International Conference on Robotics and Automation (ICRA)*. IEEE. 2008, pp. 2657–2664 (cit. on pp. 10, 20).

[60] N. Noori and V. Isler. "Lion and man with visibility in monotone polygons". In: *The International Journal of Robotics Research* 33.1 (2014), pp. 155–181 (cit. on pp. 10, 20).

[61] R. Nowakowski and P. Winkler. "Vertex-to-vertex pursuit in a graph". In: *Discrete Mathematics* 43.2-3 (1983), pp. 235–239 (cit. on pp. 2, 11, 20).

[62] S. Ong, S. W. Png, D. Hsu, and W. S. Lee. "Planning under uncertainty for robotic tasks with mixed observability". In: *The International Journal of Robotics Research* 29.8 (2010), pp. 1053–1068 (cit. on pp. 13, 21).

[63] *OpenCV library*. https://opencv.org/. Accessed: 2017-10-23 (cit. on p. 89).

[64] S.-M. Park, J.-H. Lee, and K.-Y. Chwa. "Visibility-based pursuit-evasion in a polygonal region by a searcher". In: *Proceedings of International Colloquium on Automata, Languages, and Programming*. Springer. 2001, pp. 456–468 (cit. on pp. 10, 20).

[65] T. Parsons. "Pursuit-evasion in a graph". In: *Theory and applications of graphs*. Springer, 1978, pp. 426–441 (cit. on pp. 2, 10, 20).

[66] J. Pearl. "The Solution for the Branching Factor of the Alpha-beta Pruning Algorithm and Its Optimality". In: *Commun. ACM* 25.8 (Aug. 1982), pp. 559–564. ISSN: 0001-0782. URL: http://doi.acm.org/10.1145/358589.358616 (cit. on p. 40).

[67] L. Pontryagin. *Mathematical theory of optimal processes*. CRC Press, 1987 (cit. on p. 14).

[68] *Qt | Cross-platform software development for embedded & desktop*. https://www.qt.io/. Accessed: 2017-10-05 (cit. on p. 53).

[69] A. Quattrini Li, F. Amigoni, and V. Isler. "Solving Pursuit-Evasion Games with Visibility in Polygonal Domains". In: *Proceedings of Autonomous Robots and Multirobot Systems (ARMS)*. 2016 (cit. on pp. 2, 3, 9, 17, 18, 20, 23, 26, 39, 57, 66, 73, 74).

[70] N. Roy, G. Gordon, and S. Thrun. "Finding approximate POMDP solutions through belief compression". In: *Journal of Artificial Intelligence Research* 23 (2005), pp. 1–40 (cit. on pp. 13, 21).

[71] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (3rd edition)*. Prentice Hall, 2009 (cit. on pp. 6, 14).

[72] J. Sgall. "Solution of David Gale's lion and man problem". In: *Theoretical Computer Science* 259.1 (2001), pp. 663–670 (cit. on pp. 7, 20).

[73] R. Stacpoole and T. Jamil. "Cache memories". In: *IEEE Potentials* 19.2 (2000), pp. 24–29. ISSN: 0278-6648 (cit. on p. 54).

[74] N. Stiffler and J. O'Kane. "A complete algorithm for visibility-based pursuit-evasion with multiple pursuers". In: *Proceedings of International Conference on Robotics and Automation (ICRA)*. IEEE. 2014, pp. 1660–1667 (cit. on pp. 9, 20).

[75] N. Stiffler and J. O'Kane. "Shortest paths for visibility-based pursuit-evasion". In: *Proceedings of International Conference on Robotics and Automation (ICRA)*. IEEE. 2012, pp. 3997–4002 (cit. on pp. 9, 20, 23).

[76] N. Stiffler and J. O'Kane. "Complete and optimal visibility-based pursuit-evasion". In: *The International Journal of Robotics Research* 36.8 (2017), pp. 923–946. URL: http://dx.doi.org/10.1177/0278364917711535 (cit. on pp. 2, 9, 20, 23).

[77] I. Suzuki and M. Yamashita. "Searching for a mobile intruder in a polygonal region". In: *SIAM Journal on computing* 21.5 (1992), pp. 863–888 (cit. on pp. 2, 9, 20).

[78] *The Computational Geometry Algorithms Library*. https://www.cgal.org. Accessed: 2017-10-04 (cit. on p. 52).

[79] B. Tovar and S. LaValle. "Visibility-based pursuit—evasion with bounded speed". In: *The International Journal of Robotics Research* 27.11-12 (2008), pp. 1350–1360 (cit. on pp. 9, 20).

[80] *Valgrind Home*. http://valgrind.org/. Accessed: 2017-10-15 (cit. on p. 54).

[81] J. Von Neumann and O. Morgenstern. *Theory of games and economic behavior*. Princeton University Press Princeton, NJ, 1945 (cit. on p. 14).

[82] E. Zermelo. "An application of set theory to the theory of chess-playing". In: *Proceedings of 5th International Congress of Mathematics, Cambridge*. 1912 (cit. on p. 14).

[83] R. Zou and S. Bhattacharya. "Visibility-Based Finite-Horizon Target Tracking Game". In: *IEEE Robotics and Automation Letters* 1.1 (2016), pp. 399–406 (cit. on p. 14).