Dynamic behaviors for stealth game characters

by

Wael Al Enezi

School of Computer Science

McGill University, Montreal

Mar, 2024

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Doctor of Philosophy

Abstract

In video games, the significance of Non-Playable Character (NPC) behavior holds profound implications as a fundamental aspect of crafting immersive and captivating virtual environments: NPC interactions play a pivotal role in shaping the overall player experience, be it venturing on quests, trading resources, or engaging in combat. Within the domain of stealth games, NPCs typically assume the role of guards, and their behavior acts as an instrumental factor that can either hinder or facilitate the player's covert endeavors. In order to accomplish this, NPCs must perform a number of complex tasks, including search, patrol, interception, and even stealthy behavior.

The individual planning tasks of NPCs in stealth games are sufficiently complex that behaviors are often scripted or hand-designed by expert game designers. One of the primary concerns is striking the delicate balance between providing a satisfying level of difficulty and ensuring the gameplay remains fair, enjoyable, and appropriately believable while being computationally feasible in a real-time game environment. Guard patrol routes, for example, should ensure sufficient coverage, and admit solutions at some level of difficulty, but must also avoid becoming overly predictable to maintain an element of surprise and challenge for players. Integrating guard patrol patterns seamlessly with level design is essential to offer players strategic opportunities and compelling stealth experiences.

This thesis presents novel techniques for efficient, dynamic generation of dynamic guard behaviors in stealth game contexts. We leverage the geometric properties of virtual environments, improving efficiency and believability by integrating the guards' sensory data into data structures that represent their coverage of the environment. We explore several design

variations, including the use of grids, straight skeleton roadmaps, and convex decomposition-based approaches to enable effective dynamic patrol and search behaviors without requiring intervention from game designers. We validate our designs quantitatively, but also through user studies that attempt to evaluate the relative difficulty and appeal of our methods on players. Our empirical experiments demonstrate that effective, dynamic NPC opponent behavior is feasible in real-time games, with different techniques generating patrol and search patterns with distinct traits in relation to coverage, computational costs, and believability.

Abrégé

Dans les jeux vidéo, le comportement des personnages non jouables (PNJ) a de profondes implications en tant qu'aspect fondamental de la création d'environnements virtuels immersifs et captivants : les interactions avec les PNJ jouent un rôle essentiel dans l'expérience globale du joueur, qu'il s'agisse de s'aventurer dans des quêtes, d'échanger des ressources ou de participer à des combats. Dans le domaine des jeux d'infiltration, les PNJ jouent généralement le rôle de gardes surveillant une zone, et leur comportement constitue un facteur déterminant qui peut soit entraver, soit faciliter les efforts de dissimulation du joueur. Pour ce faire, les PNJ doivent effectuer plusieurs tâches complexes, telles que la recherche, la patrouille, l'interception et même le comportement furtif.

Les tâches de planification individuelle des PNJ dans les jeux d'infiltration sont suffisamment complexes pour que leur comportements soient souvent scénarisés ou conçus manuellement par des experts en conception de jeu. L'une des principales préoccupations est de trouver le juste équilibre entre un niveau de difficulté satisfaisant et la garantie que le jeu reste équitable, agréable et crédible, tout en étant réalisable sur le plan computationnel dans un environnement de jeu en temps réel. Par exemple, les itinéraires de patrouille des gardes doivent garantir une couverture de l'environnement suffisante et permettre des solutions à un certain niveau de difficulté, mais ils doivent également éviter de devenir trop prévisibles afin de maintenir un élément de surprise et de défi pour les joueurs. L'intégration harmonieuse des schémas de patrouille des gardes dans la conception des niveaux est essentielle pour offrir aux joueurs des opportunités stratégiques et des expériences d'infiltration captivantes.

Cette thèse présente des techniques novatrices pour la génération efficace et dynamique de comportements de gardes dans des contextes de jeux d'infiltration. Nous exploitons les propriétés géométriques des environnements virtuels, améliorant ainsi l'efficacité et la crédibilité en intégrant les données sensorielles des gardes dans des structures de données représentant leur couverture de l'environnement. Nous explorons plusieurs variantes de conception, notamment l'utilisation de grilles, de schémas de chemin de squelette droit et d'approches basées sur la décomposition convexe, pour permettre des comportements dynamiques efficaces de patrouille et de recherche sans nécessiter l'intervention des concepteurs de jeux. Nous validons nos conceptions de manière quantitative, mais aussi par le biais d'études auprès des utilisateurs visant à évaluer la difficulté relative et l'attrait de nos méthodes pour les joueurs. Nos expériences empiriques démontrent qu'un comportement efficace et dynamique d'adversaires PNJ est faisable dans les jeux en temps réel, avec différentes techniques générant des schémas de patrouille et de recherche présentant des caractéristiques distinctes en termes de couverture, de coûts computationnels et de crédibilité.

Acknowledgements

This journey proved challenging, yet it was made possible only through the invaluable support of numerous individuals for whom I hold profound gratitude.

To my advisor, Clark Verbrugge: Your guidance, patience, and insight have profoundly influenced the course of my research and nurtured my intellectual development. I am forever grateful for your support and wisdom.

To my labmates, Dipanjan, Ivan, Adrian, Quinn, and Quentin: It was a fortunate privilege to work alongside you in the lab. I will forever treasure our discussions; you brought me more joy in being in the lab.

To my friends, Karim, Pranav, Marleen, Sara, Meray, Adel, and several others: Relocating to this city was a challenging endeavor, but you made me feel more at home. Thank you for being there; your friendship means a lot to me.

To my family: Despite the physical distance, your constant faith and trust enabled me to overcome the challenges in both my academic and personal journey. I owe a significant part of my achievements to you, and I consider myself fortunate to be blessed with you.

Lastly, I want to express my gratitude to all those whose names might not be mentioned but whose influence, whether small or large, has left a memorable mark on my journey.

Thank you all for being an integral part of this endeavor.

Wael Al Enezi

March 2024

Table of Contents

List of Figures

List of Tables

Acronyms

AGP Art Gallery Problem

AI Artificial Intelligence

FOV Field of View

GOAP Goal-Oriented Action Planning

HTN Hierarchical Task Network

JPS Jump Point Search

NavMesh Navigation Mesh

NPC Non-Playable Character

RRT Rapidly-exploring Random Tree

VisMesh Visibility Mesh

WRP Watchmen Route Problem

Chapter 1

Introduction

Sneaking past enemies unnoticed is common in popular game types like 3D action, *first-person shooters*, and *role-playing* games. Games that focus on this sneaky behavior are called *stealth games*. Stealth games are a genre of video games where players must avoid detection and sneak past enemy guards to achieve their goals. Players control stealthy characters and use tactics like hiding, silent takedowns, and disguises to complete missions without getting caught. In the simplest form, the player is tasked with navigating from one place to another in the game environment without being discovered by enemies.

Such games are exciting and immersive because they require careful planning and strategy to avoid detection. Players mainly focus on finding safe paths by avoiding patrolling guards and reaching a specific goal. Guard patrol is critical, adding depth, challenge, and realism to the gameplay. The behavior of guards and their patrol routes directly affects the player's ability to stay hidden and complete objectives covertly. Game developers typically craft guard patrol routes by placing waypoints the guards follow in a specific manner. This is a non-trivial process as developers consider the game level design, determining the areas the guards need to protect and the player needs to pass through. Playtesting is vital to fine-tuning the guard patrol and observing player behaviors to ensure an engaging and satisfying experience. The ultimate goal is to create guards challenging enough to test players' stealth skills but not overly frustrating, contributing to immersive and enjoyable gameplay.

This static approach of defining guard behaviors for each game level poses significant challenges when attempting to apply them to other levels. Each level's unique layout, objectives, and player interactions necessitate tailor-made guard behaviors, increasing development time and resources for each level. Moreover, this approach proves unsuitable for procedurally generated or player-customized environments, where predefined guard behaviors might not align with the dynamically changing layout of the game level.

Another drawback of this tailored approach is the potential oversimplification of guard behavior. Due to performance restrictions and simplicity in design, developers may restrict the guard patrol routines. This can reduce the game's overall replayability value, as players might quickly memorize guard patterns and lose the thrill of uncertainty in subsequent playthroughs. Achieving a balance between predictability and adaptability is crucial to maintaining an engaging and immersive stealth gameplay experience across various levels and scenarios.

Additionally, players often have the opportunity to engage with an ally NPC, and they naturally anticipate their ally to face similar conditions, such as being detected leads to enemy alerts. However, creating realistic stealth actions for ally NPCs proves challenging, as developers strive to avoid irritating players due to possibly unreliable comrades. The prevalent technique in mainstream games involves rendering allies invisible to enemy sensors. Although this approach lessens player annoyance from incompetent allies, it undermines player immersion when they realize their comrade remains unseen by enemies even after being spotted. A potential solution to address both concerns lies in enhancing the covert pathfinding capabilities of allied NPCs, thereby providing a more immersive and enjoyable stealth experience for players.

Implementing dynamic stealth behaviors offers several advantages, including reduced development time and costs. Game developers can leverage this approach to craft engaging and varied stealth-based behaviors, such as patrol patterns, search routines, and covert pathfinding. Furthermore, dynamic stealth behaviors serve as valuable tools for prompt

testing during game level design, enabling developers to assess and fine-tune the challenges presented to players, ensuring a satisfying and appropriately balanced gaming experience.

This thesis explores the feasibility of employing computational techniques to generate dynamic behaviors in stealth game scenarios, eliminating the need for human intervention. By employing diverse methods to abstract the game level layout, we introduce several approaches for dynamic patrol and search behavior for guards, leveraging the game level's structure. The effectiveness and appeal of these methods are evaluated through two user studies, assessing their enjoyability and level of challenge. Furthermore, we explore potential techniques to enhance player satisfaction even with relatively simple behaviors. Lastly, we develop and implement a novel stealthy path planning method, simulating the player's role in identifying a covert path while navigating the game level. This method can help in creating more reliable Non-Playable Characters (NPCs) as companions and in testing the difficulty of guard behavior.

1.1 Contributions

Within the realm of gaming, stealth covers a wide range of concepts. A common core feature in stealth games is how the enemy patrols the space to challenge the player's task of remaining hidden. However, when the player is spotted, many games end the game with failure, but most will result in an enemy chase. At that point, as the player hides, the enemy starts searching to find them again. This search behavior is essential in providing the player with another layer of challenge. On the other side, in many stealth games, players may have companion NPC, and to improve their immersion and cooperation with them, they are expected to be reasonably reliable in being hidden from the enemy.

In this thesis, we delve into three core features of stealth games. We investigate the intricacies of crafting dynamic guard patrols, their search behavior after detecting the player, and the art of stealthy pathfinding around guards with non-static patrol routes.

1.1.1 Dynamic Guard Patrol

Stealth games present the challenge of avoiding detection by patrolling guards who search for intruders. Designing guard motion patterns is complex, especially for procedurally generated levels, where hard-coding routes can limit replayability and design flexibility. In our work, we try to answer the following questions. **What are the possible ways to produce real-time dynamic patrol behavior, and how different are they in terms of efficiency and computational cost? Do these behaviors contribute to the overall enjoyment of the game?, and What general traits in patrol behavior affect player enjoyment?** Part of this work was published in two conference papers at Artificial Intelligence and Interactive Digital Entertainment (AIIDE) 2020 and 2023 [3, 6].

1.1.2 Guard Search and Dialog

When spotted, players may have to avoid guards and hide, either to achieve game objectives or to evade combat. After the player breaks the line of sight, guards start a search behavior, often relying on simplistic strategies that can be perceived as unfair or unrealistic. Guards may possess complete information about the player's position despite hiding or exhibiting artificial search patterns undermining their intelligence. To improve that, developers may use simple techniques, like predefined dialog lines. The questions we aim to answer are: **Is it possible to create a feasible, credible, and interesting but still solvable dynamic guard search behavior? Can players distinguish complex search tactics, and how does their perception of this relate to presentation components, like simulated spoken interactions from the NPCs?** This resulted in two conference papers at Conference on Games (CIG) 2021 [4], and at Artificial Intelligence and Interactive Digital Entertainment (AIIDE) 2023 [7].

1.1.3 Stealthy PathFinding

Stealth or covert navigation is a common requirement in various action and role-playing games, where players must traverse game levels while avoiding detection by enemy agents. While this obligation mainly rests on the player, finding a computational solution becomes essential for game design, testing, and enhancing the immersive experience provided by stealthy companions or other non-playable characters (NPCs) that require an algorithmic approach. Our work aims to answer the following questions. **How can we create stealthy pathfinding accommodating guards with non-deterministic patrol patterns? What are the different aspects of the game level or guard team that can affect the success of this method?** The findings from this research were presented in a conference paper at the ACM SIGGRAPH Conference on Motion, Interaction, and Games (MIG) [5].

1.2 Paper Contributions

This thesis resulted in five conference papers, where I was the primary author for each paper, and my advisor, Clark Verbrugge, provided editorial and presentation assistance. The following publications:

[3] Al Enezi, W., and Verbrugge, C. Dynamic guard patrol in stealth games. In Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (2020), vol. 16, pp. 160–166.

[6] Al Enezi, W., and Verbrugge, C. Evaluating player experience in stealth games: Dynamic guard patrol behavior study. In Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (2023), vol. 19.

[4] Al Enezi, W., and Verbrugge, C. Skeleton-based multi-agent opponent search. In 2021 IEEE Conference on Games (CoG) (2021), IEEE, pp. 1–8.

[7] Al Enezi, W., and Verbrugge, C. Investigating the influence of behaviors and dialogs on player enjoyment in stealth games. In Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (2023), vol. 19.

[5] Al Enezi, W., and Verbrugge, C. Stealthy path planning against dynamic observers. In Proceedings of the 15th ACM SIGGRAPH Conference on Motion, Interaction and Games (2022), pp. 1–9.

1.3 Outline

This thesis is constructed as follows. Chapter 2 reviews background material about games, the main actors within games, their decision-making, and pathfinding techniques. Chapter 3 describes our work in creating and empirically evaluating dynamic patrol for multi-agent scenarios. After that, in Chapter 4, we describe our user study in evaluating the user experience in playing against these methods. In Chapter 5, we expand some of our methods to cover multi-agent search for an adversary by utilizing the level layout to guide the search efforts. As a follow-up, in Chapter 6, we further expand on that behavior by coupling it with two dialogs, one with contextual information and one without. We describe our user study to evaluate the impact of dialog type on the user experience when playing against the search behaviors. After that, we focus on the other side of the stealth scenario. In Chapter 7, we detail our method for planning stealth paths against guards with non-deterministic motion in explored environments. In Chapter 8, we review the previous work relevant to this thesis in literature. Finally, in Chapter 9, we describe our conclusions and outline future work.

Chapter 2

Background

The research documented in this thesis spans various computer science domains, including artificial intelligence, robotics, and planning—additionally, geometry from mathematics.

In this chapter, we begin with a brief overview of games, providing definitions of combinatorial and video games, as discussed in Section 2.1. Following that, in Section 2.2, we present the various types of virtual characters that inhabit games. Subsequently, we delve into two fundamental aspects of these characters. Firstly, in Section 2.3, we explain the techniques employed to enable characters to make decisions within games. Secondly, we describe the mechanisms by which characters navigate the game environment, moving from one location to another, as outlined in Section 2.4. Lastly, in Section 2.5, we clarify the *straight skeleton*, which is the essential concept in geometry that serves as a foundation for our research.

2.1 Games

A game is an activity guided by principles or rules. It requires skill, knowledge, or luck where one or more individuals compete or collaborate to achieve a particular objective [113]. Games have been extensively studied and analyzed in various aspects. Combinatorial games, which are deterministic in nature, have received the most attention in literature [86]. In

contrast, video games are a relatively newer form of games that offer various designs and play experiences.

2.1.1 Combinatorial Games

Combinatorial games are turn-based and perfect information games, where all players have complete knowledge of the game state and all possible moves available. They also have clear and well-defined rules and goals and are typically designed with balance and fairness. Examples of combinatorial games include chess, checkers, Go, Nim, and Tic-Tac-Toe.

These games are commonly played between two-player games by sequentially taking turns making actions until the game reaches a terminal state [50]. These games are typically deterministic, which means the outcome depends solely on the current position of the game and the moves made by the players, without any element of chance involved. Additionally, combinatorial games typically have a finite number of states, and each state can be reached by one or more exact sequences of actions taken by the players.

The structure of these games makes it possible to evaluate the possibility of a player winning according to the game state. These games have been extensively studied in mathematics and computer science, and researchers have developed various techniques and algorithms to analyze their properties and strategies for optimal play [36].

2.1.2 Video Games

Video games differ from combinatorial games in several aspects. They are digital, interactive experiences played on electronic devices, offering immersive graphics, typically real-time presentation, and various input methods. Video games provide dynamic interactivity, ranging from simple to complex gameplay mechanics, often involving elements of chance or randomness.

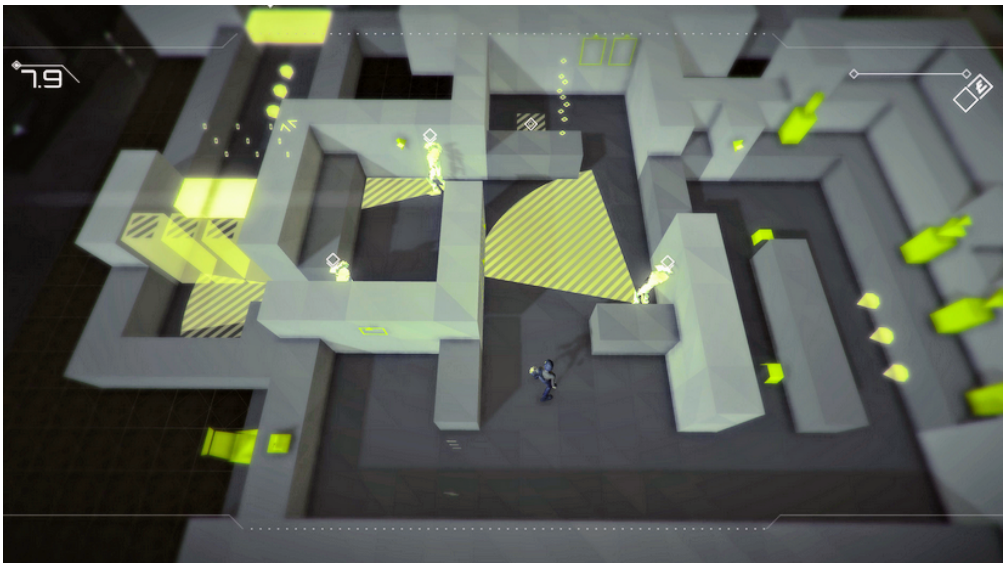
In general, video games or computer games provide a means of amusement that permits a user to engage with a simulated world through an electronic medium, such as a gaming console or a personal computer. The users, generally called players, operate input to control

one or more entities in the virtual setting to achieve one or more objectives providing players with entertainment and challenge [113].

Games can vary in the type of challenge presented to players, and therefore, they can be classified into various genres. A game genre serves as a collective term for games that share similar characteristics in terms of player perspective, limitations, or nature of the challenge. For instance, games that test a player's reaction time are categorized as *action games*, whereas those that immerse the player in a storyline are known as *adventure games*. Additionally, games can be classified according to the player's perspective, such as *first-person games*, which enable players to experience the game world through the eyes of the entity they control [123]. On the other hand, games that provide a visual viewpoint rendered from a fixed distance at the back and slightly above the controlled entity are referred to as *third-person games* [37].

Defining and classifying various genres of computer games is an ongoing area of research. This thesis, however, centers on the *stealth genre*. In *stealth games*, the player takes charge of a character whose objective is to reach a specific location within an environment without being detected by enemies. This constraint is an optional feature in several games, like Tenchu [1], where the player takes the role of a ninja tasked with infiltration missions behind enemy lines. Other games force players to adopt a stealthy approach like the game *Volume* [33]. Figure 2.1 shows a screenshot of these two games.

(a) Tenchu: Stealth Assassin [1]. The player character is shown hiding behind a wall from a non-suspecting guard.



(b) Volume [33]. The player character (dark-colored) avoids the guards(white). Their senses are modeled as a Field of View, represented in bright stripes.

Figure 2.1: Screenshots from two *stealth games*.

In game worlds, characters commonly engage with each other, and they can be broadly classified into two categories: playable characters, controlled by a player, and NPCs controlled by a form of Artificial Intelligence (AI). The interplay between these two types of characters constitutes the fundamental gameplay of many games.

2.2 NPC

NPCs are characters that players cannot directly manipulate. They exhibit certain actions that influence how players approach the game. Consequently, NPCs can be categorized into three primary roles within a game [75]:

- Opponents: The most prevalent role of NPCs in games is to provide a direct challenge to the player. For example, opponents in stealth games can be guards that the player must avoid and hide from, or, in combat games, the characters the player must defeat.
- Allies: They are NPCs that can either aid players in attaining their objectives, such as teammates in sports games, or present an indirect challenge, like a companion requiring protection.
- Background characters: These NPCs are distributed throughout the game world, like pedestrians, merchants, or villagers. These characters exhibit no inclination to support or impede the player's advancement toward their objective.

In general, NPCs exhibit specific behaviors intended to serve a particular purpose within the game, such as attacking or aiding the player. Additionally, they may display neutral actions that contribute to the player's immersion and the appearance of their intelligence, such as the ability to converse with each other or displaying natural-looking animations like a guard smoking a cigarette or yawning. The behavior and interaction of NPCs with the player is typically controlled by an AI.

The concept of game AI encompasses a wide range of functionalities that empower agents to navigate the virtual world, execute specific animations, and primarily, make informed

decisions about which actions to take. Generally speaking, the AI model can be represented and formalized as illustrated in figure 2.2.

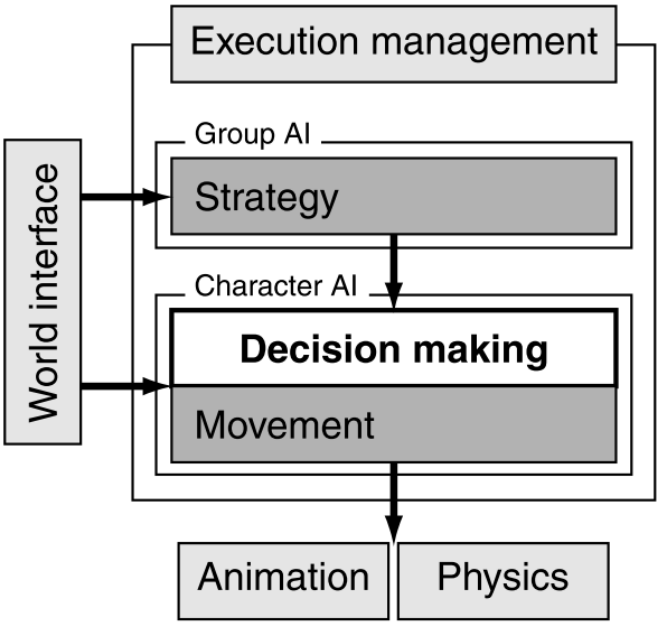


Figure 2.2: The AI model as described by Millington in chapter 5: Decision making. [78]

In the game world, NPCs rely on the **world interface** to gather information about their surroundings, such as their location and the player's health status. This information is then processed by the **Execution management** system. Game AI can be structured into multiple levels, with each level controlling either a single NPC (**Character AI**) or a group of NPCs (**Group AI**). The latter can develop a strategy for a group of NPCs to follow, such as the "Commander AI" in Killzone 3 [104]. **Character AI** defines an individual NPC behavior, or in the context of group AI focuses on lower-level actions, such as planning a path for an NPC to reach a specific location on the game level. The decisions made by the AI are then translated into animations or motions that adhere to the game world's rules and physics.

The primary objective of game AI is to enable NPCs to exhibit a semblance of intelligence, thereby presenting players with an adequate level of challenge, immersion, and en-

tertainment [20]. To accomplish this goal, game developers employ various decision-making systems.

2.3 Decision-making

Decision-making refers to selecting an action from a range of available options for an NPC, which could involve attacking a character, seeking cover, searching, or performing an animation, among other things. The purpose of each action that an NPC performs should be to achieve one or more objectives, such as challenging the player by attacking them, seeking cover when under attack from the player, or using scripted animations to enhance the realism of the NPCs.

Generally, decision-making or AI in video games have relatively unique objectives. It should be sufficiently effective, so with prolonged interaction with the player, it can provide a sense of intelligence. At the same time, it is expected to be efficient to run in real-time, and finally, it should be easily tunable and debuggable.

The decision-making process in games can be broadly classified into two approaches: *reactive* and *goal-based*. *Reactive AI* involves an agent responding to the environment by following a predetermined set of rules. On the other hand, *goal-based* behavior involves the agent seeking out the optimal action to achieve a specific objective and then determining the sequence of actions required to accomplish that goal.

2.3.1 Reactive Behavior

In this behavior, the designer controls the agent's actions to respond to specific environmental stimuli [130]. The most basic version of this approach involves conditional statements, which can be represented as decision trees.

Decision Trees

Decision trees involve an agent following a hierarchical structure to determine its actions based on observations from the environment. At each internal node of the tree, a decision is made based on the agent's observation, directing the agent to one of its children nodes, which can either be another decision node or an action to execute. The decision-making process begins at the root of the tree and concludes at a leaf node, which represents an action. For example, a simple enemy AI in a combat game can be modeled using a decision tree as illustrated in figure 2.3. Depending on the enemy's health status, their actions will vary. If their health drops below 20%, they will attempt to heal themselves using a first aid pack if one is accessible; otherwise, they will seek shelter for safety. Conversely, if their health exceeds 20%, they will initiate an attack against the player.

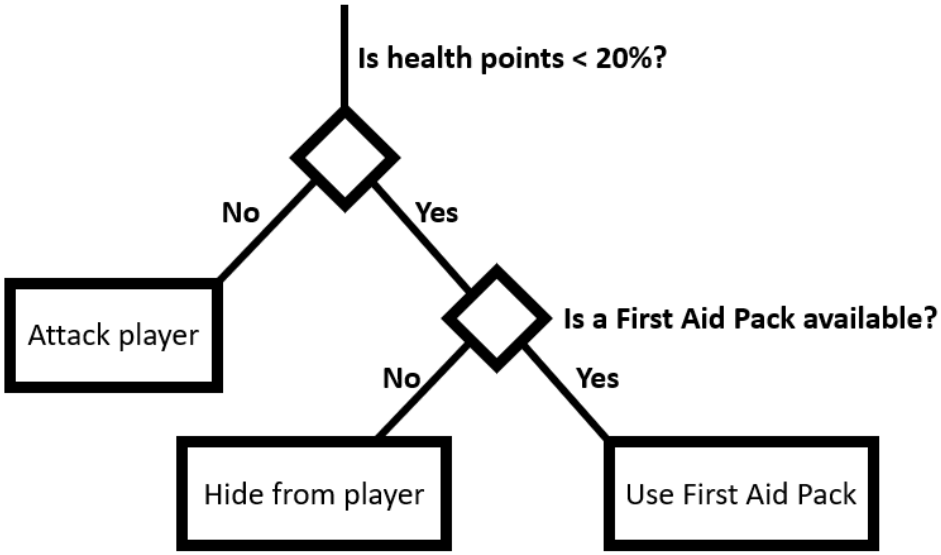


Figure 2.3: A decision tree example of an enemy behavior in a combat game.

Decision trees are known for being quick, simple, and easy to read. They can also be constructed incrementally, adding additional decisions as the agent is tested in the game. The new decisions are nested under existing nodes as child nodes, and the corresponding actions are linked to them.

State Machines

State machines are an alternative decision-making structure used in games, particularly for designing AI where an agent maintains a specific action until a trigger or event causes a change in behavior [46]. In the context of games, these state machines are commonly known as Finite State Machines (FSMs).

Finite State Machine are usually depicted using directed graphs, where nodes represent different states that contain information about specific tasks, such as attacking or patrolling. The edges in the graph indicate transitions between these states. Transitions are determined by specific conditions that, when fulfilled, cause the agent to transition to a new node. While the agent is in a given state, it performs a sequence of actions specified for that state.

During execution, the agent's current state is monitored, and potential transitions from that state are continuously evaluated to determine whether any of them are triggered. The agent immediately moves to the next state when a transition is triggered. For instance, in a stealth game, we can use this method to create enemy AI. The enemy starts by patrolling the area, and when they spot an intruder, they begin chasing them. If the intruder disappears from sight, the enemy will search until they find them again and resume the chase. This behavior is illustrated in figure 2.4. The black dot is the initial state the NPC starts from at the beginning of the game.

Designers are often drawn to using FSMs for agents due to their simplicity in design, implementation, and debugging. However, as the complexity of the behavior increases, the FSM graph becomes more difficult to modify, particularly when managing a large number of nodes and transitions. Another method can be used to model transitions between a finite set of tasks using a tree hierarchy.

Behavior Trees

Behavior trees (BTs) offer a modular approach to designing AI in games, which makes it more accessible for non-programmers to create agent behavior using a graphical user interface.

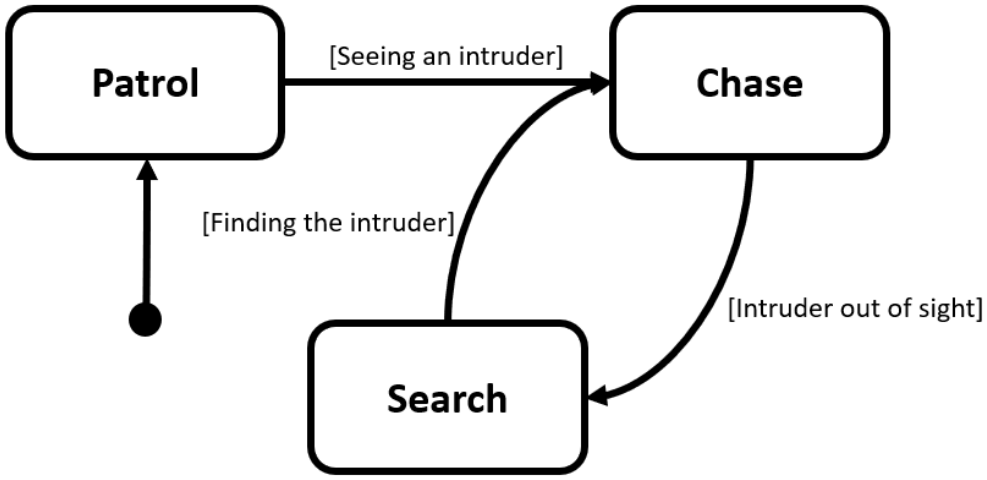


Figure 2.4: A finite state machine example for an enemy AI in a stealth game.

Unlike FSMs, BTs are composed of behaviors rather than states [130]. These building blocks, called tasks, are organized into sub-trees to achieve more complex behaviors. Each task has a designated result, typically success or failure.

In BTs, the leaves can be either conditions or actions. Conditions are nodes that evaluate whether a condition is true in the game, such as the agent's health being below a threshold or an enemy being visible. Actions are nodes that impact the game state, such as animations, pathfinding, or item usage. The internal nodes of BTs are composite nodes, which combine the previous conditions and actions into sub-trees to form more complex behaviors. There are two primary types of composite tasks: *selector* and *sequence* tasks. If we redesign the example illustrated in figure 2.3 into the format of a behavior tree, we can modify it as shown in figure 2.5. In general, every time a decision needs to be made, the execution starts from the root node.

This sample demonstrates the various types of tasks and composite nodes in a behavior tree. As can be observed, composite tasks always serve as the internal nodes of a behavior tree, while actions and conditions are the leaves set up in a fixed order. The node represented by a circle with a question mark is a *selector task*, which executes the tasks under it sequentially. If any of the underlying tasks are executed successfully, the *selector* returns a

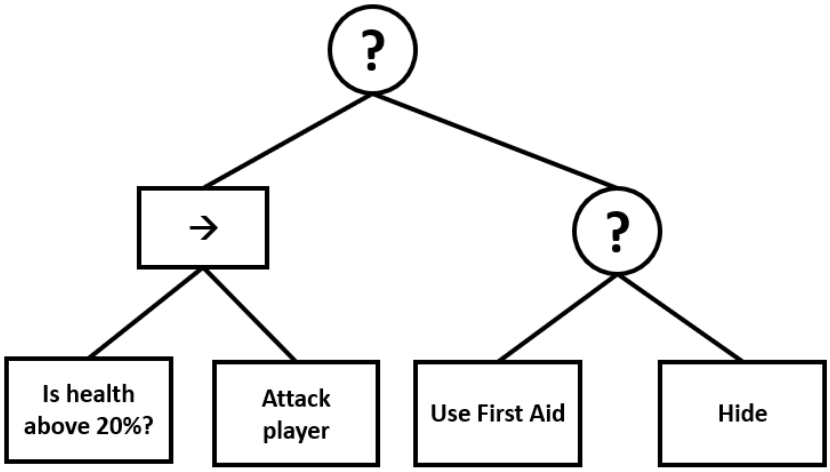


Figure 2.5: An example of Behavior Tree for the example in figure 2.3

flag indicating successful execution. The *sequence* task, represented by a rectangle with a horizontal arrow, requires all of its child tasks to be successfully executed for it to return a success flag. If any of the child tasks fail, the *sequence* task will return a fail flag.

Behavior trees can be extended by adding more elements to increase their functionality. However, since BTs do not possess state-based representation, implementing the agent to respond to external events can be challenging. This complexity arises due to the need for more nodes to enable the agent to be interrupted by external triggers.

While reactive behaviors focus on responding to stimuli conditions, they tend to necessitate a complex model to enable the agent to reason and plan for achieving a long-term objective. This issue is more effectively addressed through the use of goal-based behaviors.

2.3.2 Goal-based Behavior

Behavior that considers a particular goal or desired outcome is known as Goal-Oriented Behavior. In this type of behavior, the agent is responsible for achieving a goal or a set of goals by selecting from a range of available actions and executing the most promising one. However, as game situations become more intricate, choosing an action from a set may not lead to convincing or reasonable behavior. In such situations, there may be a need to devise a sequence of actions that would reasonably enable the agent to achieve a goal or a set of

goals. In game design, tree search, Goal-Oriented Action Planning (GOAP) and Hierarchical Task Network (HTN) have traditionally been utilized to tackle this issue [88].

Tree Search

Although a human can design an agent's behavior, it is also feasible for the agent to autonomously generate a broader range of possible states by using algorithms that consider the current state of the world and the available actions.

This approach creates a tree-like structure in which each node represents a possible state, and the edges represent the possible actions that can be taken. For instance, in a game of chess, the positions of the pieces constitute a world state that serves as a node in the tree, and the available actions represent the branches. As a result, the AI system becomes a search problem where the agent must identify the most efficient sequence of actions to achieve a desired state.

There are two types of search algorithms: uninformed and informed. In an uninformed search, the algorithm lacks knowledge of the goal, and examples include depth-first search and breadth-first search. On the other hand, an informed search involves an algorithm that has an understanding of the goal state. One of the most popular informed search algorithms is A* [53].

Although A* has demonstrated its usefulness as a search method, it becomes increasingly exhaustive and demanding as the agent's available actions expand, and the game states become more complex. Consequently, this hinders the agent's performance since it requires significant time to search for the optimal plan. For example, in the game of Go, each state has an average branching factor of 300, making it challenging for A* to be effective. To solve this problem, a new search method called Monte Carlo Tree Search was developed, which has significantly enhanced the agent's performance in Go games [22].

Monte Carlo Tree Search The MCTS algorithm has proven to be effective in tackling the issue of searching through games that have a high number of possible moves. Rather

than exploring all potential paths, it focuses on identifying and exploring the most promising ones. To accomplish this, the algorithm employs a random gameplay strategy from a given path to the game's endpoint, keeping track of whether the result is a victory or a loss.

MCTS works by simulating multiple random playthroughs to estimate the value of different actions and guide the search toward promising branches. It builds a tree structure representing the possible actions and their outcomes, expanding and updating the tree as the search progresses. Through a combination of exploration and exploitation, MCTS focuses on areas of the search space that have shown positive outcomes in previous simulations. By iteratively expanding the tree, simulating random playthroughs, and updating the statistics of visited nodes, MCTS gradually converges towards more informed decisions. The algorithm balances exploring unexplored paths and exploiting promising actions, allowing it to find good solutions in large and uncertain decision spaces. The algorithm functions by repeatedly executing four key stages in a continuous loop, illustrated in figure 2.6.

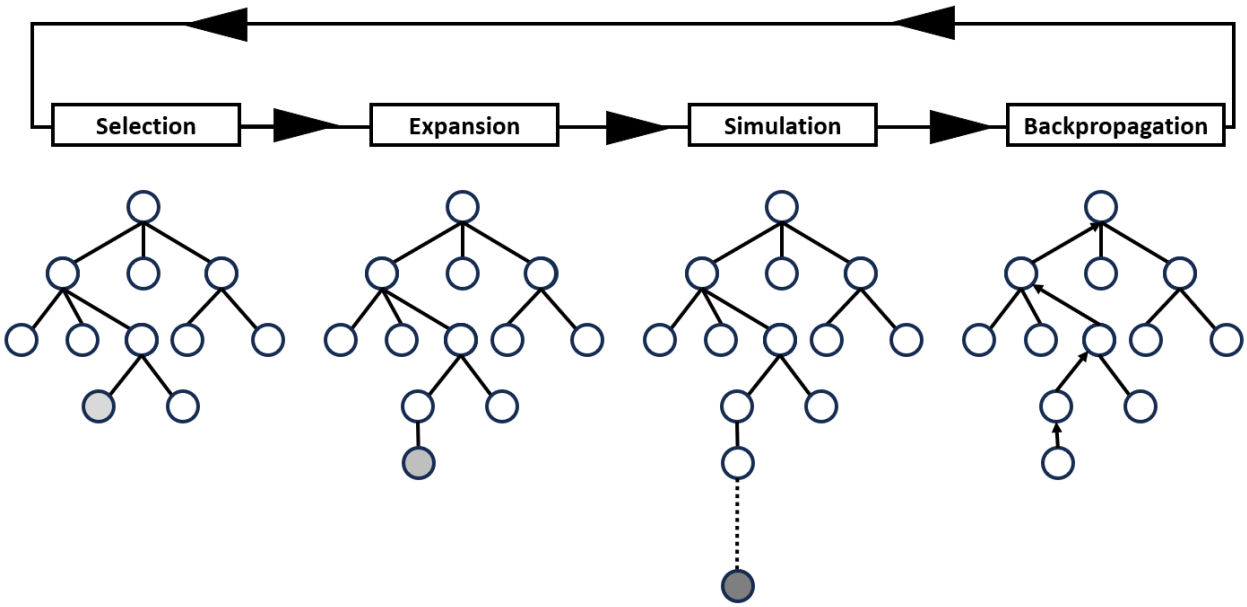


Figure 2.6: An overview of MCTS.

1. **Selection**: An unexplored node is chosen and traversed from the root node until a leaf node is reached. The selection of a path to an unexplored node is guided by the Upper

Confidence Bound 1 (UCB1) search heuristic, which is computed using formula 2.1 for the node i.

$$\frac{w_i}{n_i} \pm c * \sqrt{\frac{\ln N_i}{n_i}} \tag{2.1}$$

In this context, w_i is the number of wins possible from node i, and n_i signifies the number of times node i has been visited; by using these two values, we get the win rate of node i. c is a constant that can be tuned to encourage the selection to explore new nodes or exploit promising nodes. N_i is the number of times the parent of the node i is visited in previous iterations.

2. **Expansion**: Once a node has been chosen, it is marked for expansion, and the algorithm randomly selects one of the available actions to create child nodes.

3. **Simulation**: Once a new node has been expanded, the simulation begins by randomly executing actions until a terminal node is attained. The outcome is recorded for the terminal node, so if it is a win, w will be $+1$; otherwise, if it is a loss, then it will be -1.

4. **Backpropagation**: The results of the simulations are propagated up the tree to update all parent nodes up to the root. This process involves recording the win rate to inform future selection steps.

Although random rollout simulations offer an impartial estimation of the game state, the algorithm may not always reach a terminal state within a reasonable timeframe. In such situations, utilizing a state evaluation function with a restricted number of simulation steps can prove useful.

MCTS has demonstrated its effectiveness, and its forward sampling feature has allowed the agent to mimic human-like behavior by selecting the most promising lines of play. However, the method faces challenges when confronted with graphs with high branching factors and depths, particularly in real-time video games that require a systematic approach for

incorporating knowledge to constrain the subtree to be explored [22]. Nonetheless, MCTS has excelled in real-time video games with limited action sets, such as controlling the protagonist in Pac-Man, as evidenced by its first-place finish in the 2012 IEEE Conference on Computational Intelligence and Games (CIG'12, Granada, Spain) [93]. As game complexity increases, however, MCTS's limitations become more apparent. For instance, in Real-Time Strategy (RTS) games such as "StarCraft", MCTS may underperform due to the game's intricate nature and MCTS's reliance on randomness for rollouts to evaluate preferable future states [26, 87].

Goal-Oriented Action Planning (GOAP)

Game AI has incorporated various techniques from the field of robotics to generate intelligent behavior that appears realistic [24, 51, 126]. One such example of a technique derived from robotics is GOAP, which is founded on the STRIPS planning architecture [43]. It enables agents to create a sequence of actions on the fly to attain a goal.

GOAP involves an agent specifying a particular goal or set of goals it wishes to accomplish. The agent then searches through a set of actions to develop a plan using a planner, which it subsequently executes to bring itself closer to its goal. The fundamental components of GOAP are:

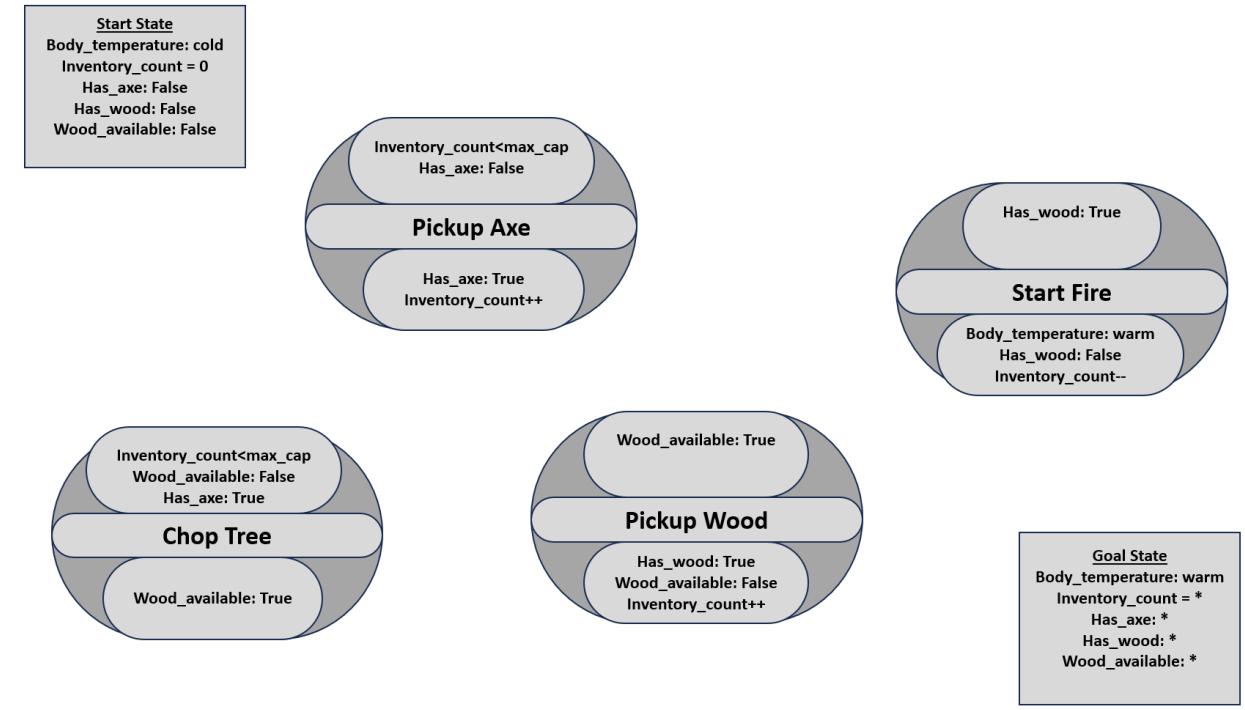
- **World State:** The state of the world is typically presented as a vector of values that describe various components of the world's condition, such as the distance to the goal position or the enemy's health.
- **Goals:** The central aspect of GOAP is a goal or a set of goals. They define the high-level intentions or tasks the agent should plan and act upon. Goals are typically represented as predicates in the world state.
- **Actions:** A set of actions is assigned to the agent, with each action containing preconditions that indicate the conditions necessary for executing the action and effects that determine how accomplishing an action will impact the state of the world. These

preconditions and effects are presented in relation to the world state. Additionally, each action can have an evaluated cost, which the planner considers when selecting an action. Each action with its preconditions and effects is encapsulated to ensure that they are considered independently of other actions, making adding or removing actions into this system simpler.

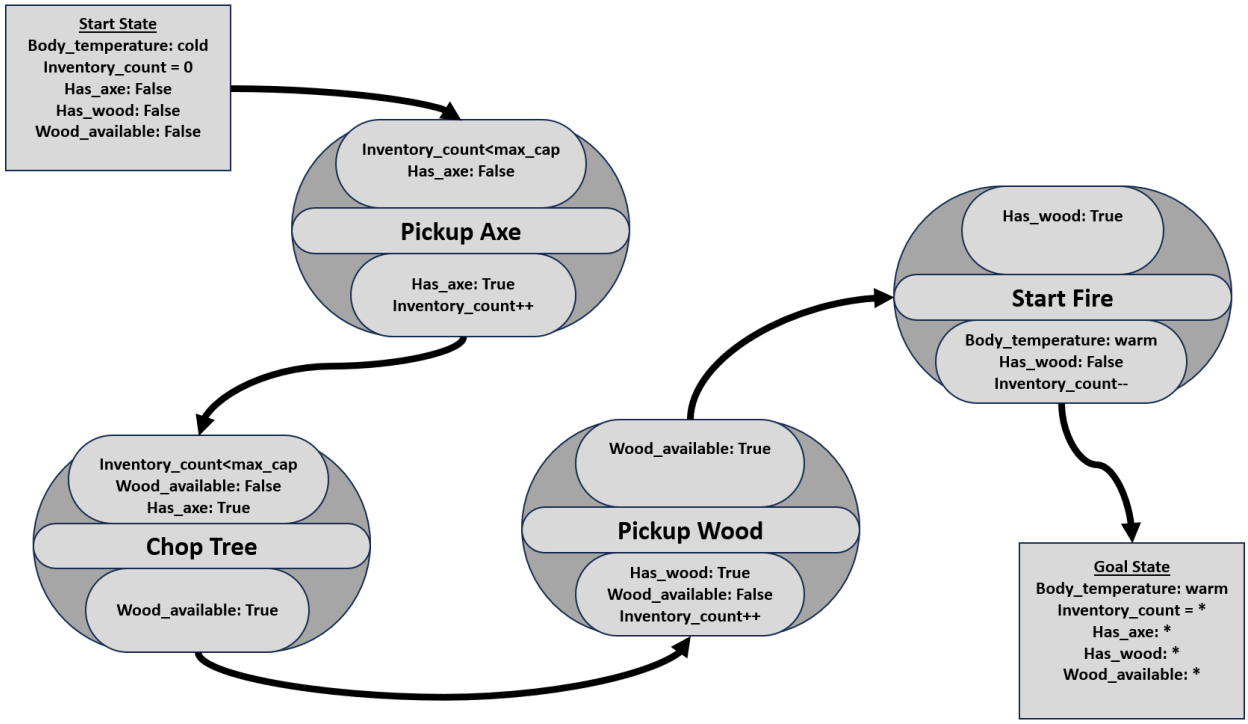
- **The Planner:** To generate a plan, the planner assesses the state of the world. This state is represented as a vector that can be compared to the preconditions and effects of the actions. The planner explores the actions to develop a plan that leads to the desired goal state. In contrast, a brute-force search through the actions can be utilized; employing a more targeted approach, such as A* is generally preferable. Using a targeted search requires the creation of a reliable heuristic that directs the search toward more promising states of the world.

- **Plan Execution:** Upon creating a plan, the agent commences its execution, requesting a new plan once the entire plan is executed or if one of its actions fails by failing the preconditions check. If a plan fails, the planner records the present state of the world as a vector and employs it as the root node to establish a new plan.

Figure 2.7 presents an illustration showcasing the structure of a GOAP. In this scenario, an NPC's current state is depicted, reflecting their feeling of coldness, while their desired state is to feel warm. To accomplish this objective, the planner examines the available actions and determines a sequence of actions that will lead to the desired state. Each action has its own set of prerequisites and effects on the world state. These actions are independent of each other, lacking any defined connections. The planner explores these actions to discover a plan that enables the agent to change the world state to the goal state of attaining a warm body temperature. The process begins with the agent selecting and picking up an axe, followed by using it to chop down a tree. Subsequently, the agent retrieves the wood and utilizes it to start a fire, ultimately resulting in the agent's warmth.



(a) The start and goal states are in rectangles, and actions are shown in round rectangles.



(b) The planner found a plan. It is shown as a directed graph so the NPCs can follow the specified sequence to reach the goal state.

Figure 2.7: An example of AI GOAP system for an NPC to warm itself in a cold environment by following the steps to start a fire.

As a result of the dynamic nature of the environment, frequent replanning is necessary due to the continuous alteration of the world state. This leads to a greater computation requirement when compared to BTs and FSMs. However, this approach results in an agent behavior that is more realistic in its attempts to achieve its predetermined goals without necessitating the creation of a complex BT or FSM. The modular structure of GOAP allows for greater flexibility in employing various agent types with distinct attributes to attain a goal with reduced development overhead. This is accomplished by defining a distinct set of actions for each agent type. For instance, in an action game, all guards can attack or move, but only the stronger guards can lift objects or obstacles. As such, stronger guards have additional actions available to them.

Hierarchical Task Network (HTN)

GOAP generally utilizes A* search or brute-force approach to determine the set of actions to be taken to achieve the goal. Due to the planner's frequent re-execution of this search process, the computational expense of maintaining the replanning task increases.

HTN employs a top-down forward-decomposition search to generate a plan, resulting in a faster planning method. The hierarchical action structure facilitates this process.

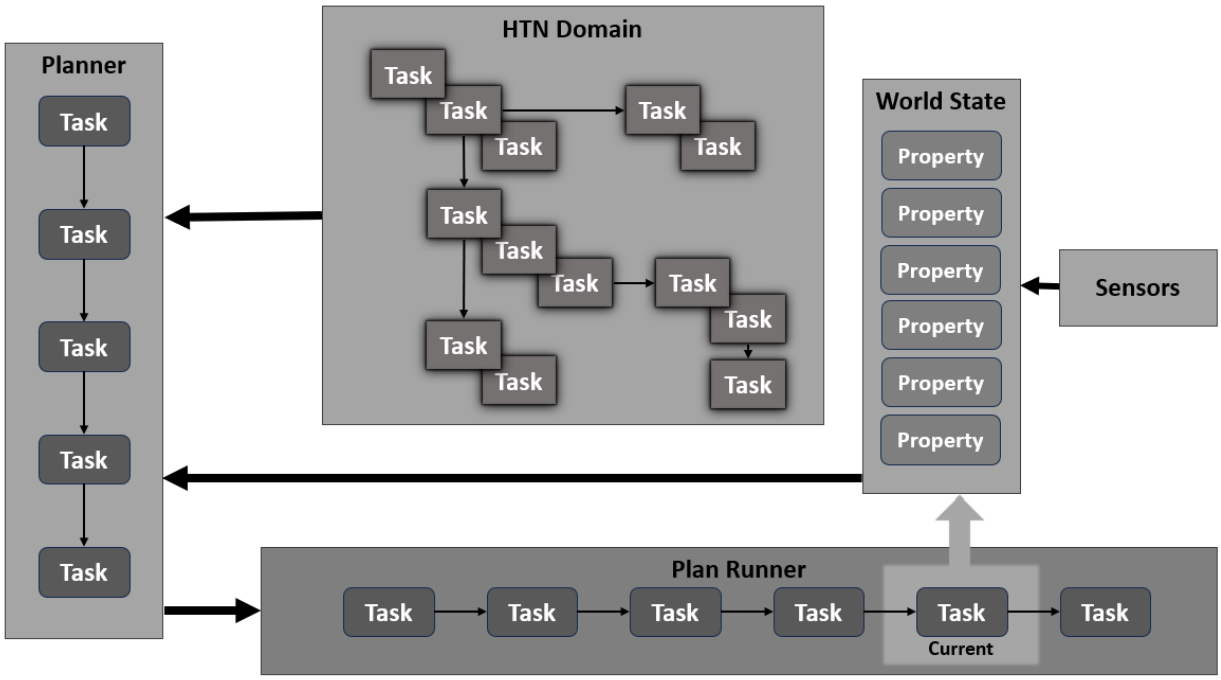


Figure 2.8: An overview of HTN.

The main components of HTN are illustrated in figure 2.8. The agent is fed with data from sensors to represent the world state. The planner then traverses the HTN Domain in a depth-first search to create a sequence of tasks. To assign a task, its preconditions are compared with a copy of the world state. The effects of the task are applied to that world state copy. Finally, the planner runner executes the tasks.

- **World State:** The agent captures and interprets information from the environment through its sensory architecture, which enables it to form a representation of the world state. Similar to GOAP, the world state is expressed as a vector of properties.

- **HTN Domain:** The hierarchy of actions the agent can follow is represented by the HTN Domain, which comprises compound and primitive tasks. *Compound tasks* refer to a task that can be accomplished by different methods, each of which includes preconditions and a list of sub-tasks. These sub-tasks may include either *compound tasks*, *primitive tasks*, or a combination of the two. On the other hand, *primitive tasks* pertain to a task with preconditions and effects on the world state once executed.

- **The Planner:** The planner's objective is to generate a sequence of primitive tasks for the agent to execute. When the agent initiates the planning process, it begins at the root node. It performs a depth-first search to construct the plan, recursively choosing composite tasks while ensuring their preconditions are met. The preconditions are verified by comparing them with the encoded world state, and the expected world state is updated based on the outcome of the set of primitive actions that each composite action reduces to. Therefore, the expected world state is compared to the preconditions of the next task. Several types of planners affect how a plan is constructed; the most commonly used ones are SHOP [82] and SHOP2 [83].

- **The Plan Runner:** Once the plan is formalized, the plan runner executes the list of tasks while checking that each task's preconditions are still met. However, if an action fails during execution, the planner is requested to replan, and the process starts again from the root node of the HTN domain. If the agent does not reach the goal state after executing the plan successfully, the planner creates a new plan to follow.

HTNs have improved planning performance over GOAP due to the efficient search enabled by the task hierarchy. However, the domain hierarchy for HTNs is built manually, unlike GOAP, where the actions and goals are listed, and the planner searches for the optimal plan. The hierarchical search for HTN allows the planner to create partial plans in the event of frequent game state changes, where plans are scrapped. Instead, partial plans can be executed, reducing the overhead in replanning and improving performance.

Utility Based System

Utility theory is not exclusive to games but is widely utilized across various fields, such as game theory and economics. The fundamental idea behind utility theory is that every feasible action or decision in a given model can be expressed numerically as a utility value, which reflects the degree of desirability of the decision in a particular scenario. For instance, when an enemy agent is low on health, they may assign a higher utility value to taking cover

rather than attacking the player to sustain themselves. Conversely, an enemy agent with full health may prioritize attacking, assigning it a higher utility value, as they can afford to lose more health [47].

Usually, the utility is normalized over the available options. This is to make it easier to compare the utilities of these options. Generally, the action with the highest expected utility is chosen. This is known as the principle of maximum expected utility [97].

Our work focuses on two primary structures: finite-state machines and utility-based systems. These two are relatively simple and fulfill the planning criteria outlined in this thesis. Finite state machines are useful for designing AI in which an agent remains in a particular state until a trigger or event prompts a transition to a different state [46]. In contrast, utility-based systems assign a numerical score to each available action at a given moment and then select one of the actions with the highest score.

2.4 Path Planning

Path planning typically finds the shortest path to a position on the goal map while avoiding collision with any obstacles; however, other restrictions can be enforced, like reducing exposure to enemies in shooter games [21]. NPCs accomplish path planning in two main steps. In the first step, they create a simplified graph of nodes and edges corresponding to the game level. In the second step, they construct a path on that graph and translate it to the game level.

The graph creation depends on how the traversable space is represented in the game. In Section 2.4.1, we describe two of the most common space representation methods. After abstracting the game level into a graph data structure, a search algorithm is applied to the graph to generate a path for the NPCs to follow. Section 2.4.2 describes the major search algorithms.

2.4.1 Space Representation

To effectively represent a game level for pathfinding, the map geometry, and the NPC's movement properties are encoded as vertices and edges of a graph. The process of associating the graph with the game's continuous world is known as localization [78]. The resulting graph will be used for the NPC's path planning. The main representation methods are as follows.

Grids

In this method, the game level is represented as a grid of nodes in 2D. Every node is assigned to a position in the game level and connected to the adjacent nodes on the grid. However, as game levels tend to include obstacles or holes, nodes that fall in these positions are considered untraversable and have no edges connected to them. Figure 2.9 shows an example of a grid; The hollow dots represent traversable nodes, while the black nodes are non-traversable.

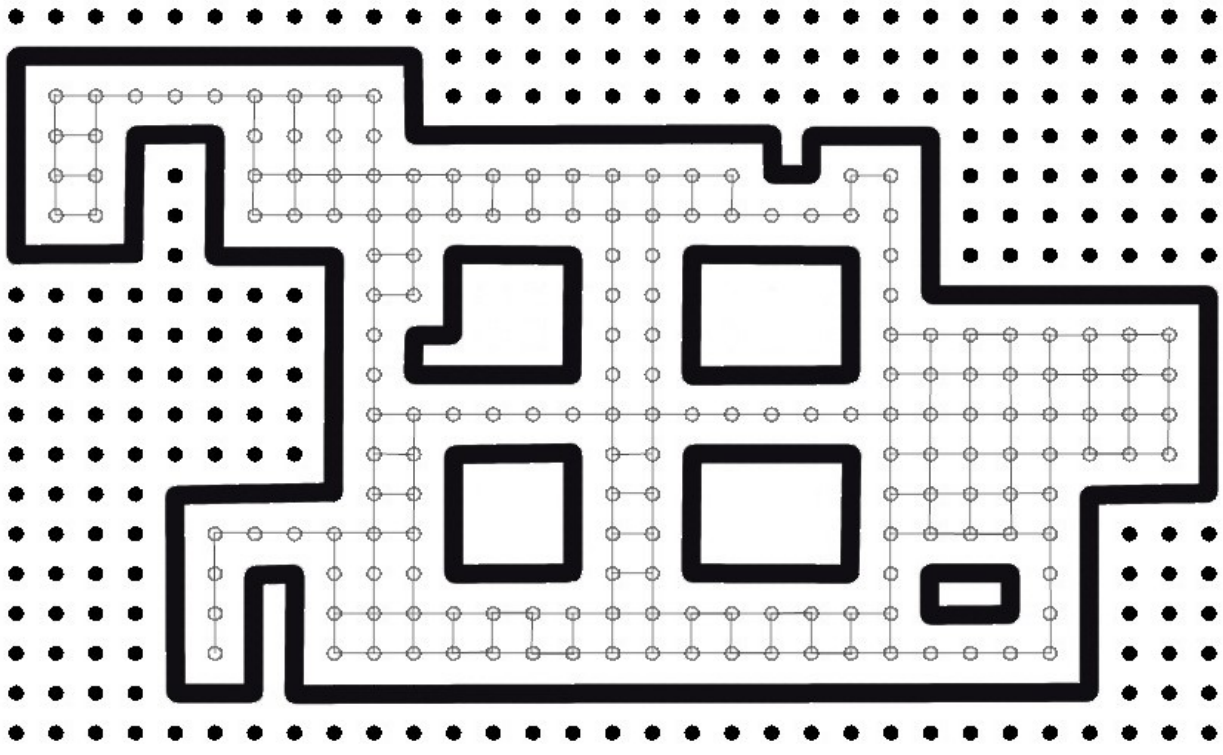


Figure 2.9: Top-down view of a map from Metal Gear Solid 1

Grid graphs may simplify the space representation task, but this method has limitations. The NPC's movement model is restricted to be either vertical or horizontal, called *Manhattan* movement, where it has 4-way movement [32]. Sometimes a diagonal movement is possible, giving the agent an 8-way movement. Such models are called *Chebyshev* or, in case of having different weights to diagonal movement, *Octile* movement. These models prevent the NPC from planning a flexible, believable trajectory and reaching specific locations not localized on the grid. However, it is possible to tune the grid to be more fine-grained to closely represent real terrain and obstacles [74]. For example, in a coarse-grained grid, a large obstacle, such as a boulder, may occupy only one cell, leading to imprecise planning. However, by tuning the grid to be more fine-grained, the same obstacle can span multiple cells, accounting for its actual shape and size, enabling the agent to plan more accurate and safer paths. Fine-grained grids may improve NPCs' mobility but will result in a larger graph, which can be expensive to compute.

Navigation Meshes

The limitations of grids can be overcome by more accurate representations of the game level. A method, initially used in robotics, created a graph free-space representation by decomposing the world's space into convex regions labeled as "meadow areas" [9]. This method is more commonly known as "navigation meshes" in video games [102].

Navigation meshes (known as navmesh) represent the game level's traversable space in convex polygons. The reason for partitioning the space to convex polygons is that if two points are on the same polygon, they can be connected by a straight line, so having these polygons as the building blocks of a graph will simplify navigation. Each polygon represents a node on the graph, and the nodes are connected if their corresponding polygons share edges. Since edge-sharing determines the connectivity of a node, the maximum number of connections a node has will depend on the number of edges in that polygon [78].

Navmesh Construction The process of setting up navmeshes is more complicated than grids. Changes in them tend to be expensive and challenging to implement due to their dependency on the world's geometry. Alternatively, they could be hand-drawn by the designer or automatically generated.

In the case of automatic navmesh generation, the initial step is to obtain a convex decomposition of the walkable surfaces in a game world. Each convex region is the building block of the navmesh. Eventually, a path can be built across these convex regions. A conceptually simple approach to convex decomposition is partitioning the space into triangles. In the case of 3D games, it is possible to obtain the walkable triangles from a given triangular mesh representation of the space by filtering the triangles with their normals facing up. However, if we only had the map's geometry, we could use various polygon triangulation algorithms. Perhaps the most straightforward approach is a brute force approach, repeatedly adding internal edges "diagonals" as long as there are no edge intersections. However, a brute-force approach with complexity $O(n^3)$ could be computationally expensive. More efficient methods use polygonal properties, such as Meister's ear-clipping algorithm [76], which takes advantage of the fact that every simple polygon of more than three vertices has at least two "ears". An "ear" is the three adjacent vertices that form an interior triangle and do not contain any other vertices. The complexity of this method is $O(n^2)$. More efficient solutions, such as plane sweep, are possible but are significantly more complex [100]. Additionally, there is even a more efficient triangulation algorithm presented with linear complexity, however, it is more complex and only theoretical [27].

The number of convex polygons determines the complexity of path planning; the fewer convex polygons there are, the simpler it is to plan a path. In the case of a triangulated space, the resulting navmesh is usually suboptimal in the sense of being over-partitioned, and the world could be represented by merging the triangles to form larger convex shapes. This will require fewer polygons to represent the game space. A common method for merging triangles is the Hertel-Mehlhorn algorithm [118]. Figure 2.10 shows an example of the

convex partitioning phases. The input is a triangulated navmesh, and after using the Hertel-Mehlhorn algorithm, the polygon is partitioned into fewer polygons.

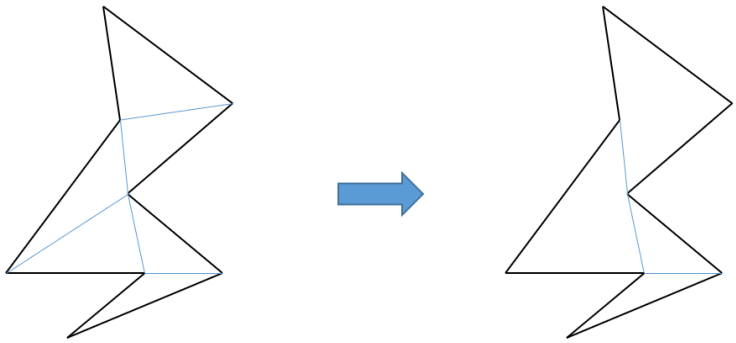


Figure 2.10: An example of a triangulated map before and after using the Hertel-Mehlhorn algorithm

The algorithm iterates over the internal edges and removes "inessential" diagonals. An inessential diagonal is an internal line between two vertices, which, if removed, the resulting polygon will remain convex. Hertel-Mehlhorn runs in linear time and guarantees no more than four times the number of optimal convex polygons.

After partitioning the space, each convex polygon is considered a node in a graph, and its edges are the links to the adjacent polygons.

The NavMesh method provides a more accurate space representation since the NPC's movement is not restricted to two or three axes, like in grids. The resulting graph of this representation is relatively sparse compared to grids. This gives a NavMesh approach a performance boost in path planning, so we need less memory and computation power.

2.4.2 Pathfinding in Graphs

After translating the game level to a graph, it is possible to create a plan for the NPC to follow to reach its goal. The process of associating an NPC's position on the game level with the graph is known as quantization. From a quantized starting point and goal location, an algorithm is run to establish a path on the graph, and then it is translated back to the game

level coordinates for the NPC to follow. The main algorithms used for pathfinding are as follows.

Dijkstra

This algorithm was originally used to solve the shortest path in graph theory [39]. This algorithm aims to find the shortest paths from a starting node to every node on the graph.

Dijkstra's algorithm search behavior starts from the initial node and spreads to the neighboring nodes while keeping a record of the shortest paths to each visited node. Dijkstra's algorithm iterates on every node, adding the adjacent nodes to a list of "to-be-visited" nodes. It updates the lowest cost from the starting node to each node. After updating all the neighboring nodes, the current node is moved to the "visited" nodes list. A new current node will be chosen, and the new current node is the one with the least cumulative cost from the start node. As the algorithm expands the search through the nodes in the graph, the goal nodes should be found, and the path to the goal is backtracked to the start node returning the shortest path. This search will return the shortest path; however, it is computationally expensive because it does not have an intuition to guide the search toward the goal. The algorithm's worst case is $O(n \log n + e)$, where n is the number of nodes and e is the number of edges.

A*

Dijkstra's algorithm may provide the shortest path to a node; however, it finds it by naively expanding its search in the surrounding space. It lacks an "intuition" to guide the search to nodes that are most likely to be on the path to the goal node. A* came up with a solution to this shortcoming. It originally came from the project "Shakey the robot" [85].

In Dijkstra's algorithm, the criteria for choosing the next current node is the cumulative cost to that node. This makes the search unguided; however, A* provides a heuristic $h(n)$ to determine the direction of its search [54]. The algorithm's performance relies on this heuristic, which estimates how far the goal node is. Equation 2.2 shows the heuristic A*

uses to pick its next "current node"; $g(n)$ is the cumulative cost to a specific node n, and $h(n)$ is the heuristic of the node. This heuristic represents the estimated distance to the goal node. The most commonly used heuristic is the Euclidean distance to the goal node.

The complexity of the algorithm depends on the heuristic it uses. The heuristic will allow it to limit the branching factor in a search to reach the goal. Given the path length is b, and the maximum number of edges per node is d, the complexity would be $O(d^b)$. A* guarantees finding the optimal solution (i.e., the path with the lowest cost) if the heuristic function is admissible (never overestimates the cost to reach the goal).

$$f(n) = g(n) + h(n) \tag{2.2}$$

Pathfinding in Navmesh After abstracting the game level into a graph, given the source and destination points, a path can be found using one of the graph search algorithms mentioned before. Each node maps to a polygon, and each pair of polygons in that path includes a shared edge that marks the transition between them. The final path is then constructed by ensuring a feasible, local path between each shared edge. A trivial approach to doing this is to connect the midpoint of each shared edge to the centroid of each of the polygons sharing that edge. As each polygon is convex, this connection is guaranteed to be obstacle-free, and by connecting the start and endpoints to the first and last shared edges, respectively, a complete path can be realized. Figure 2.11.a shows an example of a navmesh path through centroids.

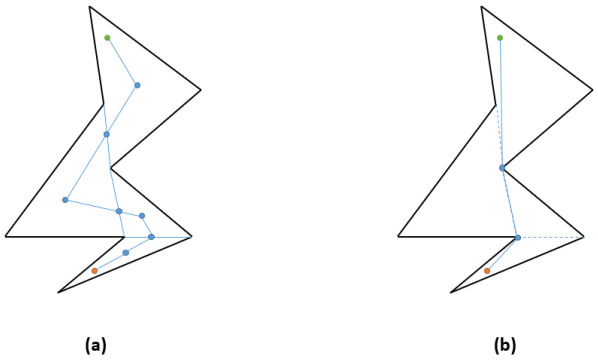


Figure 2.11: (a) A centroid navmesh path. (b) A navmesh path using the "simple stupid funnel" algorithm.

More advanced methods reduce the jaggedness in the output path by avoiding the need to connect the entry and exit of a polygon through the centroids while still ensuring obstacle edges are not clipped. An example is the "simple stupid funnel" algorithm [79]. It is based on the metaphor of pulling a piece of string internal to the polygon taught between the start and destination points. It incrementally constructs a path, creating a "funnel" based on the bounds given by the need for the string to pass through the next shared edge. Subsequently, smaller shared edges reduce this window, contracting it to a single point when the path necessarily goes around a corner. Figure 2.11.b shows the path generated for the previous example.

To better understand how it works, Figure 2.12 shows how the path was constructed using this algorithm in the context of triangulation. The funnel algorithm only considers the vertices on the inner edges. The vertices of the first internal edge mark its funnel sides, with the source point as the funnel's apex, as seen in (b). First, it checks if the left and right vertices are inside the funnel; if so, the left side is moved to the next vertex and then rechecks if both sides are still inside the funnel, as shown in (c). If so, the right vertex is moved to the next, and so on. However, if the next right vertex is to the right of the right side, it skips that vertex. For example, in (c), the right side is on vertex H, and the next is G, but G is to the right of the right side, so it skips to vertex F, which is on the left of the

right side. Once a vertex is detected outside the funnel, like in (d), the funnel arms cross one another. The vertex on the other side is added as a node to the path and marked as the new apex of the funnel. The vertices of the next internal edge are the sides of the funnel. The algorithm repeats until the destination point is reached, as shown in (g).

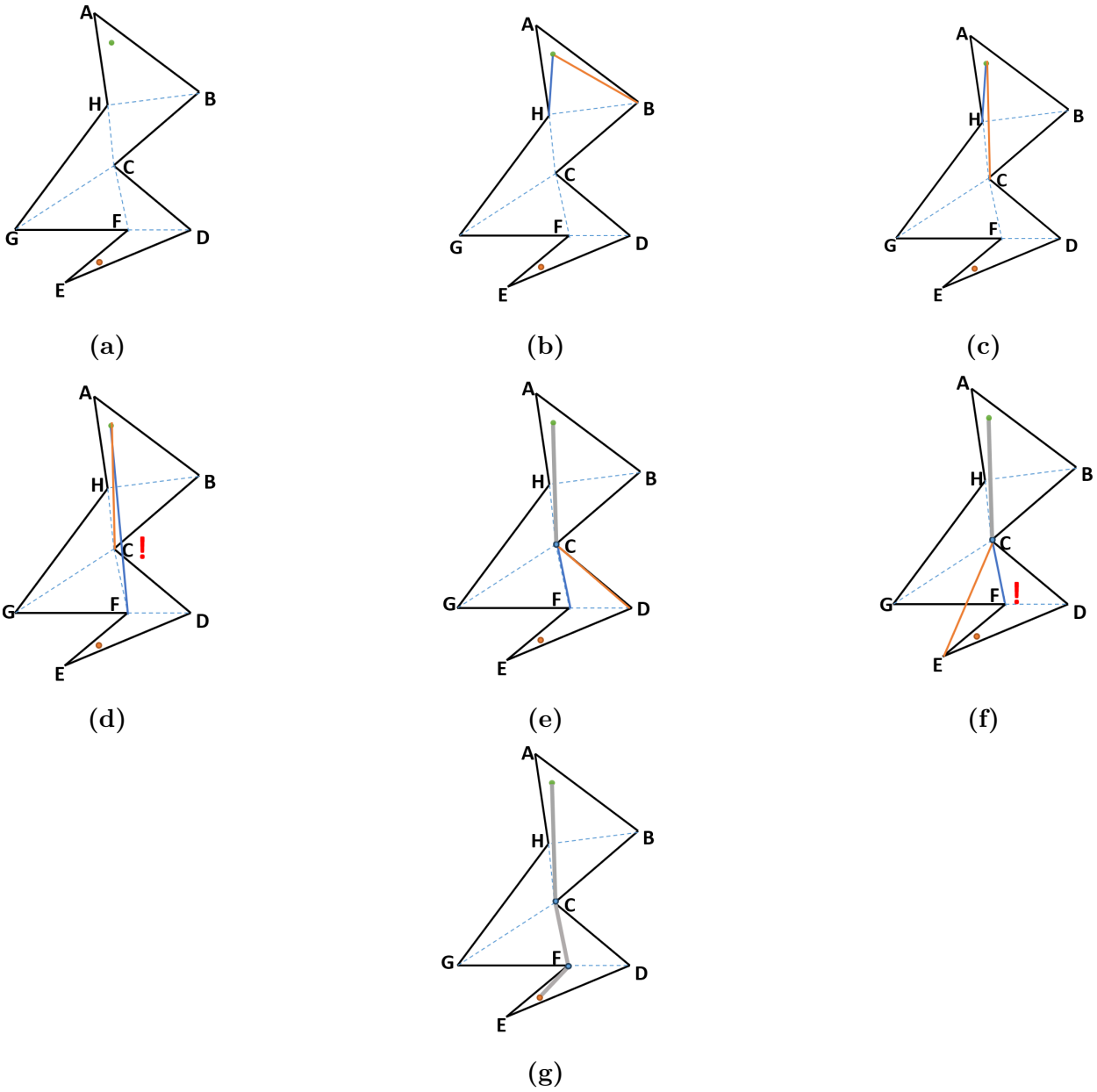
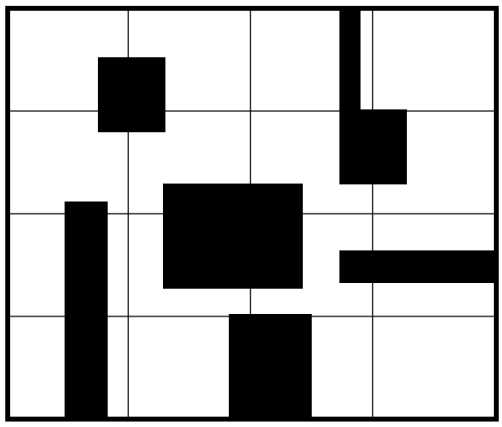


Figure 2.12: An example of how the "Simple Stupid Funnel" algorithm works.

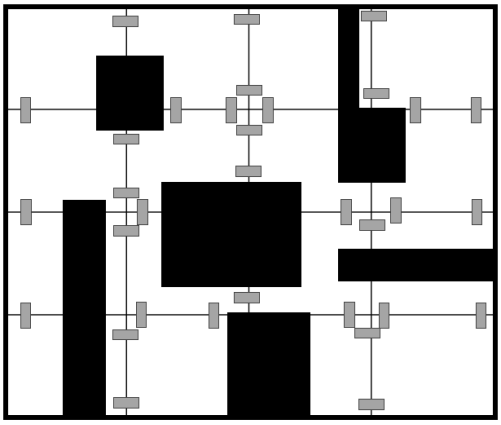
Hierarchical Pathfinding (HPA*)

This method divides the world into levels of detail by grouping original graph nodes into linked local clusters [19]. The world is represented in multiple levels of detail for the game's space. For example, an agent wants to go from one room inside a building to another in a different city. First, the agent generates the overall plan of reaching the goal city and then plans the path that satisfies the overall path on the higher-level graphs. After reaching the goal city, it plans the path to the goal street and, finally, the goal building.

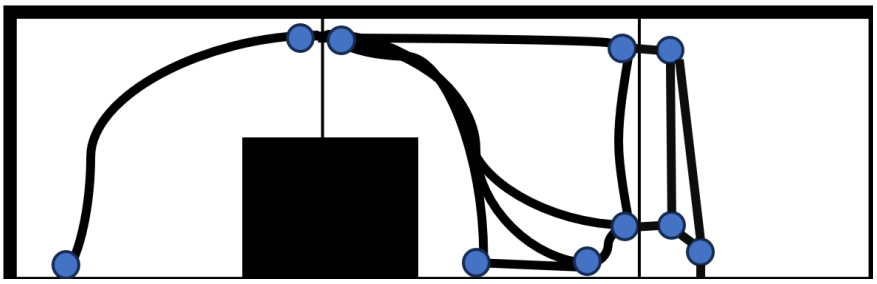
Figure 2.13 shows a top-down view of a game level with obstacles shown as black polygons. The method starts a topological abstraction from the whole level by dividing it into clusters. The grouping is done by dividing the world into rectangles; one rectangle is considered a cluster. The cluster size can, of course, be adjusted. Next, it identifies the cluster connections. A cluster connection is specified in the common border of two adjacent clusters. As a tile on the cluster border is selected, the symmetrical tile is linked to it on the other cluster. Eventually, every connection tile should be on the cluster's border and has a symmetrical, obstacle-free tile connected to it from the adjacent cluster as shown in Figure 2.13.b.



(a) The game level is divided into clusters.



(b) Game level with cluster connections shown as grey rectangles.



(c) Top 3 clusters in the game level showing the cluster connections as blue dots and the intra-edges as black lines connecting the dots. To reduce clutter, not all intra-edges are shown.

Figure 2.13: An example of hierarchical pathfinding in a simple game level.

Figure 2.13.c shows the top left 3 clusters in this example. The blue nodes represent the cluster connections. The entrances to each cluster are on its borders. Each obstacle-free border segment has an entrance or two, depending on its length. The entrances are then added as nodes along with the connection between them. Since these connections link two clusters, they are referred to as inter-edges. The length of an inter-edge is considered to be 1. The connections inside the same cluster are referred to as intra-edges. Their length is computed by the shortest path inside the cluster.

When an agent provides the start and goal nodes, they are inserted into the abstract graph. So each node is linked to all entrances of its cluster. Then A* is used to find the path between these points on the abstract graph. For cases where extra hierarchy levels

are required, multiple levels of clusters can be implemented to refine the graph further. Although HPA* decreased computation in large and complex game spaces compared to A*, it increased path jaggedness due to rerouting through cluster connections, resulting in less efficient paths.

Pathfinding can be further optimized by making assumptions about the game space. Jump Point Search (JPS) is such an approach, being one of the most efficient search algorithms. However, it also brings some caveats regarding the game space.

Jump-Point Search (JPS)

This algorithm introduced a more optimized search performance than A*. However, it can only be used in a limited world representation. JPS is restricted to work on grid-based game spaces that have uniform-weight distance cost [52]. It considers symmetry and reduces the search to more promising nodes and prunes nodes that are not part of the optimal path. In general, instead of considering exploring the eight neighbor nodes like A* does, JPS eliminates several of those nodes trivially to reduce the redundant exploration.

To understand how JPS prunes unnecessary nodes, we can simplify the possible movement on the grid as either horizontal/vertical or diagonal. In the first case, Figure 2.14 shows two possible horizontal expansion cases. The cell x is the current node, and p(x) is the node's parent. When considering which neighbors of x to expand, it pruned p(x) because it previously expanded it. In the open space pruning, JPS also pruned nodes 1 and 7 because they could be optimally reached through p(x). It also prunes 2 and 8 for the same reason; they are optimally reached through p(x) instead of passing through p(x), x, and then either of these nodes. Nodes 3 and 9 can be reached from p(x) through 2 and 8 at the same cost as via x, so we also ignore that redundant path. This leaves node 6 as the next node to expand; we refer to it as a natural neighbor. As for the case of an obstacle, Figure 2.14.b shows a case of an obstacle above x. This forces us to consider node 3 since the cheapest route to it necessarily passes through x, and it will be added for expansion later. In this case, node

3 is referred to as a forced neighbor. Other configurations of obstacles are handled on a case-by-case basis.

(a) Open space pruning **(b)** Pruning with obstacle

Figure 2.14: Vertical or Horizontal pruning

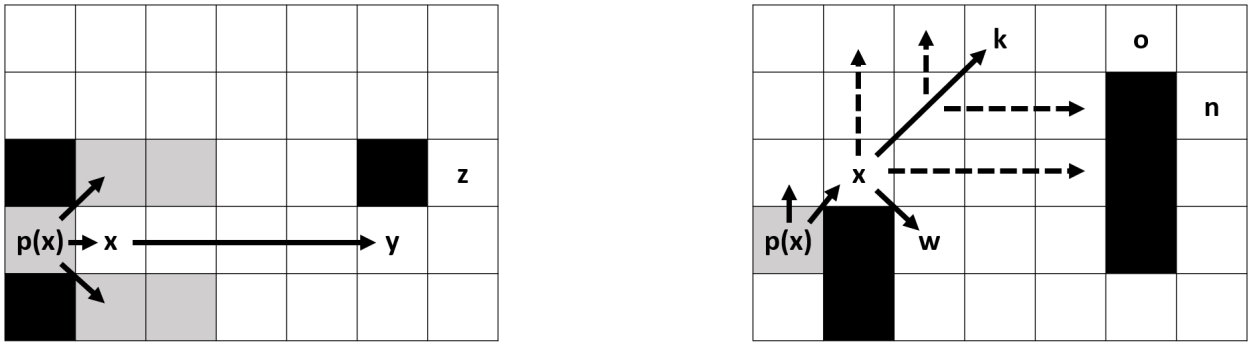
As for the diagonal movement, similar simplifying assumptions are made. Figure 2.15 shows two examples of diagonal movements. In open space pruning, nodes 4 and 8 are ignored because they can be reached optimally from p(x). The same applies to nodes 1 and 9 because they can be optimally reached without passing through x. This leaves nodes 2, 3, and 6 as natural neighbors. Here, node 6 will be expanded horizontally, 2 vertically, and 3 diagonally. The expansion is done vertically and horizontally, then diagonally. As for the presence of obstacles, similarly to the horizontal movement, the neighbor of x next to the obstacle, which is node 1, is added as a *forced neighbor*.

Figure 2.16 shows examples of how JPS considers the interesting nodes that can be part of the optimal path to a goal. In (b), we can see that nodes x, k, and o are added as *natural neighbors* since they are on the optimal path to the *forced neighbors* w and n. *Natural neighbors* are the nodes that remain after the vertical and diagonal pruning. Generally, the algorithm expands through a straight path, so at a specific node, it only considers those nodes that can be optimally reached through that node. A diagonal node that can only be optimally reached through a current node due to an obstacle is added as a *jump-point*. *Jump-points* are thus those points where the search is recursively restarted.

(a) Open space pruning **(b)** Pruning with obstacle

Figure 2.15: Diagonal pruning

JPS starts a recursion horizontally until it finds a node that should be added as a jump-point to search from, like the node y in figure 2.16.a. Node y is added as a jump-point since z is optimally reached through it. Once the recursion fails to find the goal from the current node, the search restarts from the added jump-point nodes.



(a) Horizontal case **(b)** Diagonal case

Figure 2.16: Examples of paths found by JPS. Dashed lines indicate a sequence of interim node evaluations that reached a dead end. Strong lines indicate eventual successor nodes. Adapted from [52]

JPS introduced some improvement over A* because it restricts the nodes considered in the search. However, unlike A*, its application lies only in uniform-cost grid world spaces [52].

2.5 Straight Skeleton

In games, geometry holds significance within the realm of video games. Our work incorporates a geometric concept called the *straight skeleton*. The straight skeleton is a geometric concept that describes the evolving shape of an object as it undergoes a shrinking or eroding process. It is obtained by continuously shrinking the original shape inward until it disappears while tracing the boundaries of the successive offsets [16]. The straight skeleton has a similar data structure called the medial axis. It has several distinctions, like using curves in its presentation, etc [71].

The straight skeleton can be represented as a set of line segments that connect the evolving corners of the original shape. These line segments trace the boundaries of the shape as it shrinks, capturing the topological changes that occur during the process. Figure 2.17 shows an example of a straight skeleton of a simple polygon.

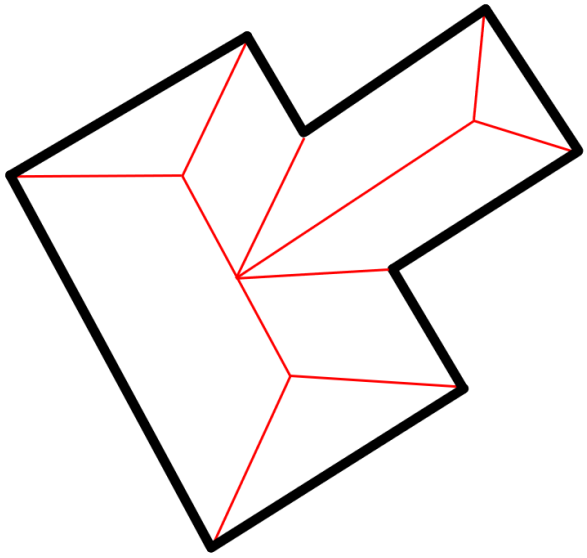


Figure 2.17: Example of a straight skeleton (graph in red) of a polygon.

Several methods can be used to draw straight skeletons [2, 28, 59]. This work uses a method that discretizes the game world into a grid [84]. This method simplifies a polygon into a binary image so that each pixel is flagged as walkable or un-walkable. The implementation

of this approach is relatively straightforward, facilitating its adoption in our work. Here is a summary of the steps involved in this method:

1. **Create Distance Transform:** The distance transform calculates the distance of each pixel in an image to the nearest boundary of a shape or object. This results in a grayscale image where each pixel value corresponds to its distance from the boundary. It resembles an elevation heatmap where the higher the distance transform of a cell, the higher its elevation.

2. **Find Skeleton Components:** Using the distance transform values, it is possible to find several essential points for constructing the skeleton, like local maximums, *saddle points*, or *humps*. Local maximums represent groups of pixels with the highest distance transform. *Saddle points* are locations within the distance transform where the distance values are neither minimal (close to the boundary) nor maximal (far from the boundary). Instead, they are situated between the lowest and highest distance values. *Saddle points* often occur at points where two or more pixels in an image are close to each other, creating a "saddle" or low point between them. They can be significant in determining the connectivity and relationships between different points. *Humps* are regions within the distance transform that exhibit elevated distance values. These regions typically correspond to protrusions or outlying areas on the object's boundary. In the distance transform, humps represent areas where the distance from the object's boundary is relatively higher, indicating that these areas are farther from the center.

3. **Connect Skeleton Components:** The skeleton components are then modeled as nodes and are connected by expanding towards other visible nodes. This creates a rough skeleton shape. After that, the final skeleton structure can be obtained by simplifying the resulting graph to essential nodes and edges.

The straight skeleton of a polygon is a two-dimensional graph that maintains the original polygon's connectivity and comprises straight line segments that link points on the polygon

or the edges of the straight skeleton itself. This structure finds various applications, such as the design of buildings [107], road networks [125], and urban planning [58]. In general, the straight skeleton of a polygon possesses numerous properties that render it useful for geometric analysis.

2.6 Summary

In this chapter, we provided the background relevant to this dissertation. We formalized the concept of games and described into the various types of NPCs that inhabit them. Following that, we discuss the techniques involved in decision-making and path planning within games. In the subsequent chapter, we explain our approach to formulating dynamic guard patrol behavior in stealth games.

Chapter 3

Guard Patrol Behavior

In stealth games, a key challenge involves evading detection by guards, who typically patrol or move through an area searching for intruders. Guard motion patterns are context-dependent and difficult to design, particularly for procedurally generated game levels. While commercial games often hard-code guard patrol routes, this approach can limit replayability and design flexibility. Therefore, dynamically generating patrol routes is a desirable solution that reduces design effort and allows for unique guard behavior in each playthrough.

Guard patrols serve as obstacles and challenges for players to overcome, adding a layer of difficulty by introducing potential threats and raising the stakes. Players are required to strategize and devise methods to navigate through or bypass patrols without being detected. The patrol routes are composed of a sequence of waypoints or areas that guards traverse. These routes are strategically designed to cover specific or significant areas within the game level, aiming to challenge players and prevent them from passing through unnoticed.

To address the challenge of designing guard patrol routes in stealth games, we formalize patrol route construction in terms of "staleness". This involves partitioning the game level into n regions or points and assigning a staleness value s_i to each region. The staleness value increases over time and is reset when a patrolling guard observes the region. The objective is to create a patrol route that minimizes overall staleness. This approach has been previously

used to measure patrol performance for robots [64]. By minimizing staleness, guards can effectively cover the game level and increase the game's difficulty for players.

In this chapter, we explore three distinct approaches to automating the construction of patrol routes for multiple agents. Our design seeks to provide guard AI with various representations of the game world, which can be achieved by modeling the relative staleness of areas in the game level. This can be done using one of three approaches: a typical discrete grid-based context, similar to the use of occupancy maps in exploration problems [24, 62, 81]; a *straight skeleton* graph-based approach that emphasizes the map's primary structure; or a mesh-based spatial decomposition. Each method has its own set of advantages and disadvantages, such as differing levels of complexity, efficiency in creating patrol behavior, and computational cost. By exploring these approaches, we aim to identify the most effective solution for automating multi-agent patrol route construction in stealth games.

We conduct experiments to evaluate our approaches to automating patrol route construction for multiple agents. We test various parameters that influence the decision-making processes with the different world representations across multiple game levels created in Unity3D. Our experimental results indicate that the approaches exhibit similar behavioral characteristics but differ in terms of runtime performance. These results illustrate that in a gaming context, the creation of efficient, dynamic patrols covering game levels can be realized through various approaches for modeling the game environment. The primary contributions of this chapter include:

- We describe three approaches to dynamic, runtime patrol route construction, using either a simple grid-based, a *straight skeleton*-based, or a more flexible space decomposition-based model. Our techniques apply to both defining patrol routes for good level coverage as well as for solving specific exploration problems.

- We empirically evaluate and compare these approaches using multiple game maps inspired by commercial video games. Implementation within Unity3D shows that all

approaches are relatively effective, but two can be efficiently realized in a real-time game environment.

This chapter is organized as follows: In Section 3.1, we introduce the standard scenario that we use to assess the effectiveness of our patrol route construction methods and the primary metric used to evaluate their performance. Next, in Sections 3.2, 3.3, and 3.4, we provide a comprehensive description of the patrol methods that we developed. Afterward, we outline our experimental setup. Section 3.5 involves setting the parameters for each method and comparing the performance of the three methods against a basic, inexpensive baseline patrol approach. Finally, in Section 3.6, we present the results of our experiments, including the patrol performance metrics and the computational resources required by each method when implemented in our system.

3.1 Scenario

To test our methods, we developed a test-bed prototype as a top-down stealth game. The level is represented as a simple polygon with holes; the interior of this polygon defines the walkable area. An arbitrary number of NPCs are spawned on the walkable area; we refer to these NPCs as guards. Each guard has a visual sensory represented as a partial disk of a fixed range and angle commonly referred to as Field of View (FOV).

The guards have a fixed movement and rotation speed to move around the game level. The patrol behavior is defined by the way these guards move to secure the game level from any unwanted entities that can pass through. Figure 3.1 shows a screenshot of our prototype developed in Unity3D. The scenario begins with the guards spawning on the game level for a set time and concludes when the designated time ends. Although there are many parameters that can affect performance, we discuss them in later sections.

Researchers use the percentage of the area covered to evaluate exploration performance in robotics [92]. Regarding patrol, previous work in robotics studied patrol performance. It used the uniform coverage of the area as a heuristic, assuming that the more evenly robots

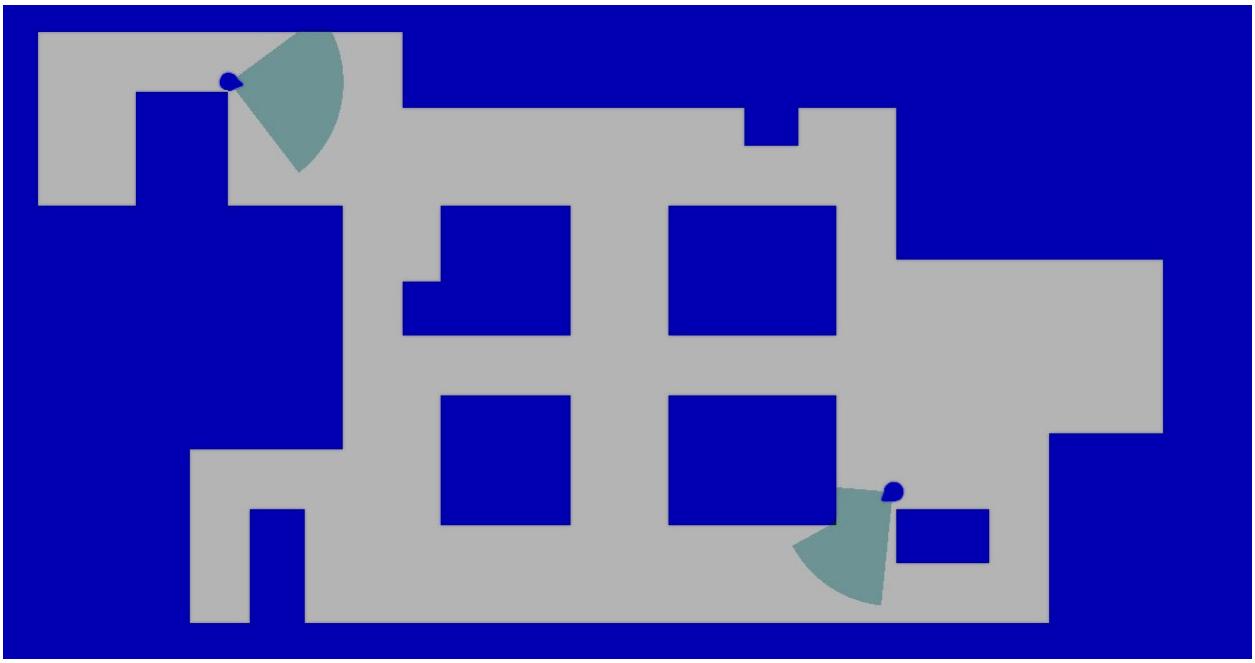


Figure 3.1: A screenshot demonstrating the prototype used in our study, as shown to players. The walkable area is grey, the guards are navy blue dots, and their corresponding FOV is the partial light blue disk. The game level layout is from the game "Metal Gear Solid" [68], the Docks map.

surveyed the area, the more effective their patrol behavior is [64]. We follow the same method to assess the dynamic patrol performance quantitatively.

In the following sections, we introduce the three patrol methods we created. Each method comprises two main steps, so during the game, the method iterates between these two steps until the end of the scenario. The first step is representation-update: this is important to reflect the changes in the world state so that as guards move, we have an up-to-date representation of the game level that considers the previous actions. The second step, decision-making, is where the guards decide to plan their paths in the game level based on the specific representation of the game level.

3.2 Grid-based

This method relies on representing the game level in a grid to track how frequently guards cover the corresponding area. To do that, we discretize the game level into a grid of cells of fixed dimensions, where each cell is either walkable or non-walkable. To measure the individual coverage of cells, we associate each cell with a numerical value that indicates the time passed since it was last covered by a guard's FOV; we refer to this value as *staleness*. For simplicity, we considered a cell to be covered once its center point is in the FOV of a guard. All guards always share the same data and use it to plan their subsequent patrol trajectory.

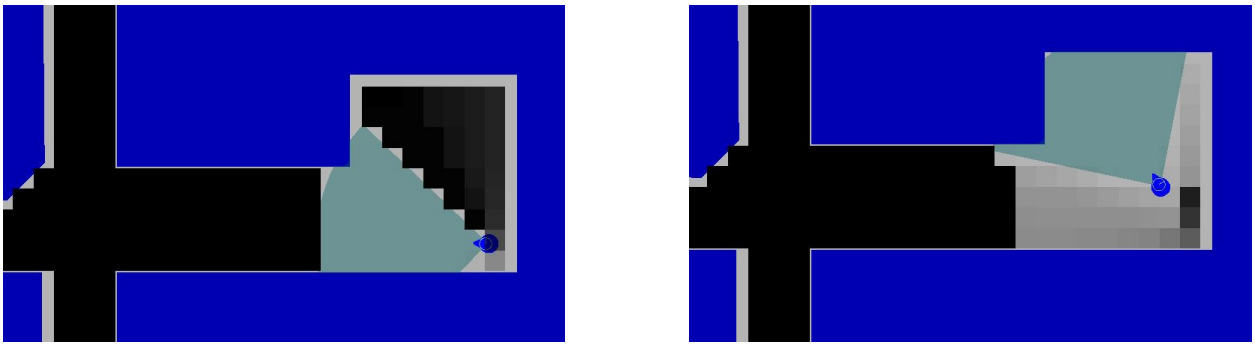
3.2.1 Representation-Update

At every time timestep, we update the cells' staleness according to algorithm 1. In general, when a cell is in one of the guards' FOV, we update the last time it is seen. Subsequently, the staleness is the normalized duration a cell has gone unseen relative to other cells. This results in a staleness value between 0 and 1, with 0 representing the most recently seen cell (depicted in a bright color) and 1 indicating the cell unseen the longest (depicted in black). Figure 3.2 shows an illustration of this representation.

Algorithm 1 Grid staleness update

Require: $N_{walkable}$, the cells in the walkable area.

for each $n \in \mathcal{N}_{walkable}$ **do**
  if $IsVisibileByGuard(n)$ **then**
    $n.lastSeenTime \leftarrow t$
  end if
end for
$t_{oldest} \leftarrow Min(N_{walkable})$
$t_{newest} \leftarrow Max(N_{walkable})$
for each $n \in \mathcal{N}_{walkable}$ **do**
  $n.staleness \leftarrow (n.lastSeenTime - t_{oldest})/(t_{newest} - t_{oldest})$
end for



(a) **(b)**

Figure 3.2: Screenshots showing how the guard's FOV affects the grid cells. The guard is shown in navy blue, and the FOV is shown in light blue. The darker the color of a cell, the higher its staleness. This illustration depicts the shortcoming of discretizing the game level so that areas are considered to be covered that are not covered since the center of the corresponding cell was covered. This can be seen as a staircase effect on diagonal grid cells.

3.2.2 Decision-Making

After the grid representation is updated, each idle guard will choose a cell to cover. Once a guard selects a specific cell, it uses the Navigation Mesh (NavMesh) to find the shortest path toward that cell. As soon as the guard surveys the designated cell, it requests a new target cell. The goal of the guards is to lower the overall staleness of the grid.

To consider other factors in choosing the suitable cell to cover other than its staleness, each guard picks their target cell based on a numerical value; we call it fitness. The fitness value is a weighted average of the corresponding cell properties, including its staleness value, path distance from the guards, and path distance from the guard itself. Equation 3.1 shows how the fitness of a cell n is calculated.

$$\begin{aligned} f(n) = n.staleness * w_{staleness} + (1 - NormalizedDistance(n, g)) * w_{distance} \\ + DistanceClosestGuard(n, g) * w_{separation} \end{aligned} \tag{3.1}$$

The function $NormalizedDistance(n, g)$ returns the shortest path distance between a guard g and cell n. In addition, the distance is normalized by dividing by the game level diameter. For efficiency, we precalculate the game level diameter by comparing the shortest path distances between the interior corners of the game level, and then we choose the longest path distance. Our calculation serves as an approximation to the longest shortest path between any two locations in the game level. $DistanceClosestGuard(n, g)$ returns the shortest normalized path distance of the closest guard to cell n, given that guard is not g. The path distances are calculated using the NavMesh. As for $w_{staleness}, w_{distance}$ and $w_{separation}$, each is a value between 0 and 1. Each weight controls how influential a property is to the overall fitness of a cell to a guard. These properties are:

- Cell staleness: The staler a cell, the more urgent it is for a guard to cover it. The corresponding weight is $w_{staleness}$.
- Path distance ($NormalizedDistance(n, g)$): This property indicates if closer cells are more desirable or vice versa. The corresponding weight is $w_{distance}$.
- Closest other guard distance ($DistanceClosestGuard(n, g)$): Improving coverage efficiency requires guards to increase the time they spend away from each other and spread across the game level. Selecting cells farthest from other guards may offer better separation and improved coverage. The corresponding weight is $w_{separation}$.

The weights serve as adjustable variables in the fitness equation and are set by the user. In section 3.5, we examine the impact of modifying the weights on the coverage performance.

3.3 Roadmap

Maintaining a grid representation incurs costs as every cell must be updated at every time step. The need for more cells increases with larger game levels, making it computationally challenging and potentially infeasible. We introduce an alternative approach that requires

fewer units for updates by using a rough approximation of the game level rather than a precise grid representation.

We simplify the game level into a planar graph representation, which we name the roadmap, by using the straight skeleton of the game level, explained in section 2.5. This abstraction allows us to reach any point on the walkable space on the game level. The intent is that covering it with guard FOVs will likely cover a significant portion of the game level. The coverage performance heavily relies on the range of the guard FOVs, and it is typical for commercial video games to set the FOV to a limited range. For our experiment, we set the guard FOVs to 10% of the longest side of the bounding box of the game level. Setting the range of the FOV as a portion of the game level dimension rather than a fixed length will scale the FOV's range according to the size of the game level. We defined this value to match the radius of guard FOV in commercial games like "Metal Gear Solid". Future tests can be done to measure the impact of how FOV's range affects patrol performance.

We further discretize the roadmap by dividing its edges into shorter segments with a fixed length so each segment can fit into a guard's FOV. Then each segment is associated with a numerical value that ranges from 0 to 1, which reflects its staleness. This reflects when a guard last covered this segment. The discretization step allows us to separate the staleness value between the different segments to better approximate the covered areas. Figure 3.3 shows an example of a game level with the roadmap rendered.

As with the grid representation, guards share the roadmap to help them decide their future moves. Similarly, the main steps of this approach are updating the presentation and making decisions for the guards.

3.3.1 Representation-Update

As the guards move throughout the game level, they aim to cover the roadmap segments, reducing their staleness, a value between 0 and 1. As the guard approaches a segment, the length of the non-visible portion of the segment decreases until it reaches 0, at which point its staleness is reset to 0 and considered covered. At each time step, the staleness of the

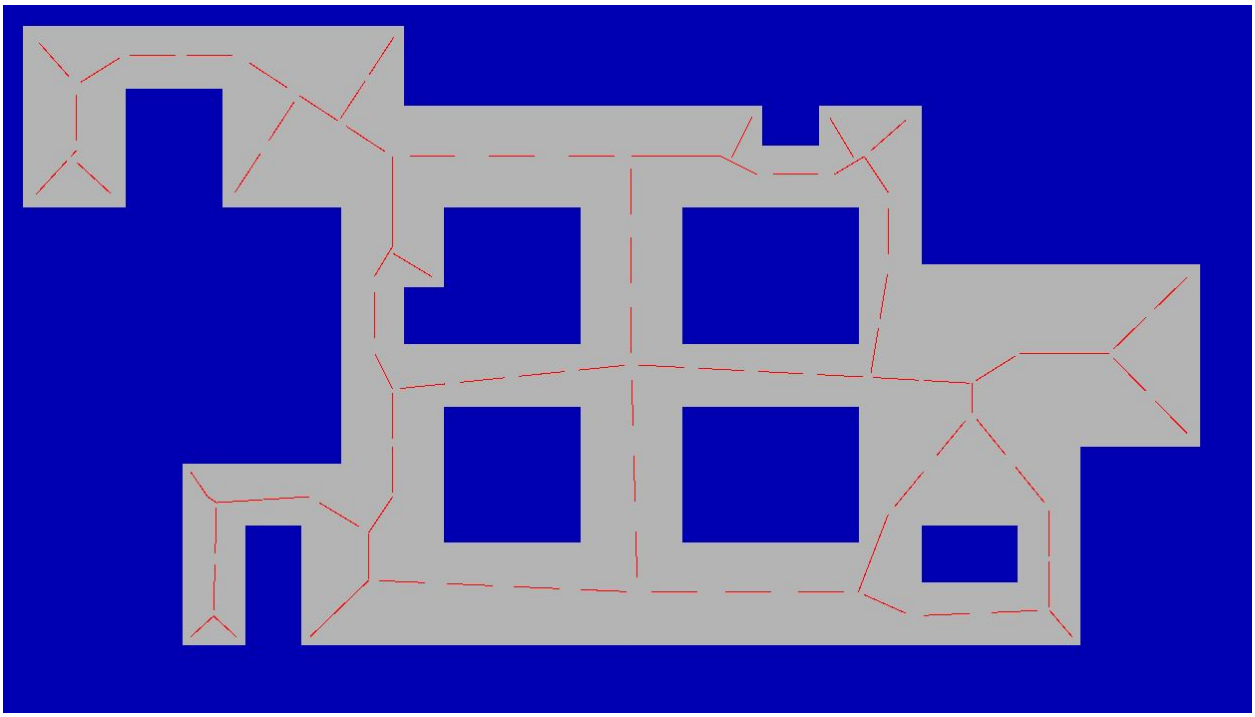
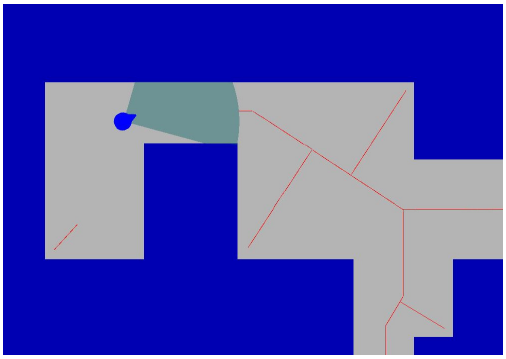


Figure 3.3: A screenshot depicting the roadmap with segments separated for clarification while their ends are connected in the implementation. The walkable area is grey, and the segments are colored red. The game level is from the game "Metal Gear Solid", the Docks level [68].

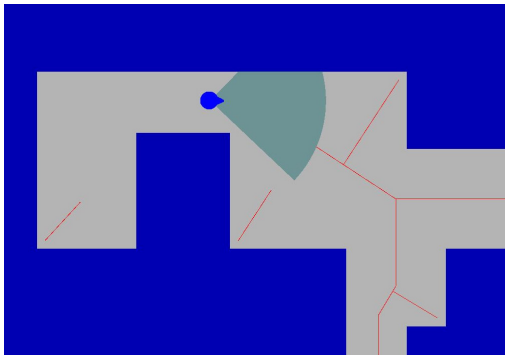
segments is increased by a fixed rate, then normalized to preserve the relative difference between them. Finally, the segments are adjusted if they intersect with or are contained within the guards' FOV. Figure 3.4 illustrates how a guard's FOV affects the segments.

3.3.2 Decision-Making

Once the roadmap is updated, any inactive guard will request a plan for their next move. In this representation, we have defined two types of plans. In the first one, the guard selects a segment and finds the shortest route to it using the NavMesh. In the second, the guard creates a complete path to cover a group of segments in sequence rather than just selecting destination segments. The motivation for the latter is to consider the path taken instead of a single segment. We first explain the first variation, where the guards simply choose a destination to cover, and then we explain the other alternative of building a complete path.



(a) The guard is moving and covering the segments with its FOV. As this is happening, the segment that partially falls in the FOV shrinks in length until it is completely covered by the FOV.



(b) Once the segment is fully covered by the FOVs, its staleness is set to 0. The screenshot shows those segments with non-zero staleness in red, and those with 0 staleness are removed.

Figure 3.4: Screenshots showing how the guard's FOV affects the roadmap segments.

Choosing a Destination

Similar to the grid representation, we evaluate the fitness of a segment by computing the weighted average of its properties and assign the segment with the highest fitness to the next available idle guard. Equation 3.2 shows how the fitness is calculated.

$$f(s) = s.staleness * w_{staleness} + (1 - NormalizedDistance(s, g)) * w_{distance} + DistanceClosestGuard(s, g) * w_{separation} \tag{3.2}$$

These properties are similar to those described in section 3.2.2.

Building a Path

In this plan, the guards plan a complete path instead of just a destination segment. Upon a guard determining a path, the relevant line segments are updated such that when other guards plan their path, they select line segments with fewer or no guards traversing through

them. This is expected to promote the separation between guards not only at the destination but along the whole path.

Algorithm 2 details how we achieve path construction for a guard. In summary, GetClosestSegmentOnRoadMap(g, S) returns the closest segment in the roadmap to guard g (line 9); It represents the start segment of the potential path to be built. After that, similar to Dijkstra's shortest path algorithm, we build a path by exploring the path with the highest total utility by expanding out from the guard's location. However, we stopped expanding the search if the total distance reached the defined limit (lines 13-15). We expand the search by iterating through the connected segments to the current segment and update the possible highest total utility it can reach along with the total distance to reach it (lines 16-23). After the search is over, we backtrack the path from the segment with the highest utility to the start (lines 30-32).

As the acquired path follows the roadmap, we refine it by removing extra nodes and optimizing going around corners using the NavMesh while keeping its overall abstract trajectory. We explore the impact of setting the maximum search distance in section 3.5.

A guard will select a path for their next plan based on its overall utility, which is the sum of individual utilities of the segments along that path. The utility of a segment "$GetUtility(s)$" is the weighted average of its properties. This method relies on the values of the following properties and global variables:

- MAX_DISTANCE: This is the max length of the constructed path. This is usually set as a percentage of the game level diameter.
- Segment staleness: The staleness of a line segment refers to the duration since any of the guards have covered it. A value of 1 denotes that it is the oldest line segment that is yet to be covered, while a value of 0 signifies that it has just been covered.
- Guards portion: For every segment, we will save the count of guards that intend to cross it. We standardize this count by dividing it by the number of guards in the game level.

Algorithm 2 Roadmap decision making for a guard

```
1: function GetPath(S,g) where S is the group of segments in the roadmap, g is the guard requesting a path.
2:     open_list ← {}
3:     best_segment ← null
4:     for Every segment s in S do
5:         s.path_utility ← 0
6:         s.path_parent ← null
7:         s.total_distance ← 0
8:     end for
9:     n ← GetClosestSegmentOnRoadMap(g, S)
10:    open_list.enqueue(n)
11:    while open_list is not empty do
12:        c_s ← open_list.pop()
13:        if c_s.total_distance >= MAX_DISTANCE then
14:            break
15:        end if
16:        for Every non-visited neighbour segment s of c_s do
17:            utility ← GetUtility(s)
18:            total_utility ← utility + c_s.path_utility
19:            if total_utility > s.path_utility then
20:                s.path_utility ← total_utility
21:                s.path_parent ← c_s
22:                s.total_distance ← c_s.total_distance + s.length
23:            end if
24:            open_list.enqueue(s)
25:        end for
26:        if best_segment.path_utility < c_s.path_utility then
27:            best_segment ← c_s
28:        end if
29:    end while
30:    if best_segment is not null then
31:        return GetSimplifiedPath(best_segment)
32:    else
33:        return null
34:    end if
35: end function
```

- Connectivity: This represents the count of segments that are linked to a specific segment. This attribute may prioritize paths that traverse highly connected segments. To standardize this value, we divide the number of connected segments by 10, which we consider the maximum count of connections that a segment can have according to our preliminary testing for the game levels we considered.

- Passing guards threshold: It's a binary attribute that influences $s_{guardPortion}$. The first option is "ACTUAL", in which $s_{guardPortion}$ holds its value as defined. The second choice is "MAX", where $s_{guardPortion}$ is set to 1 when one or more guards pass through and 0 when no guards are crossing the segment.

Equation 3.3 shows how the utility of a segment is calculated. The weights "w" for the corresponding properties are values set between 0 and 1. We explore the impact of these weights in section 3.5.

$$GetUtility(s) = s_{staleness} * w_{staleness} + s_{guardPortion} * w_{guardPortion} + s_{connectivity} * w_{connectivity} \tag{3.3}$$

3.4 Space Decomposition (VisMesh)

The roadmap results in fewer units requiring updates compared to the grid representation. This is by simplifying the game level into a simpler graph, meaning that covering the graph may provide less coverage of the game level than with the grid representation. In this section, we define a method that aims to achieve a low count of units to update while maintaining a complete representation of the game level.

This method relies on utilizing the geometry of the game level and decomposing it into smaller units that get covered as the guards move around. As the guards patrol, the space is further decomposed to reflect the changes caused by their motion. To understand this better, we explain the components of this approach:

- Game level region: The game level is represented as a simple polygon with holes. Guards move inside the interior while aiming at covering the walkable space.

- Covered region: As guards move through space, we geometrically union the set of polygons formed by their FOV. The covered region represents the exact area guards have covered since a point in time. Figure 3.5 shows an example of a guard moving in the game level and expanding the covered region as a result of this motion.

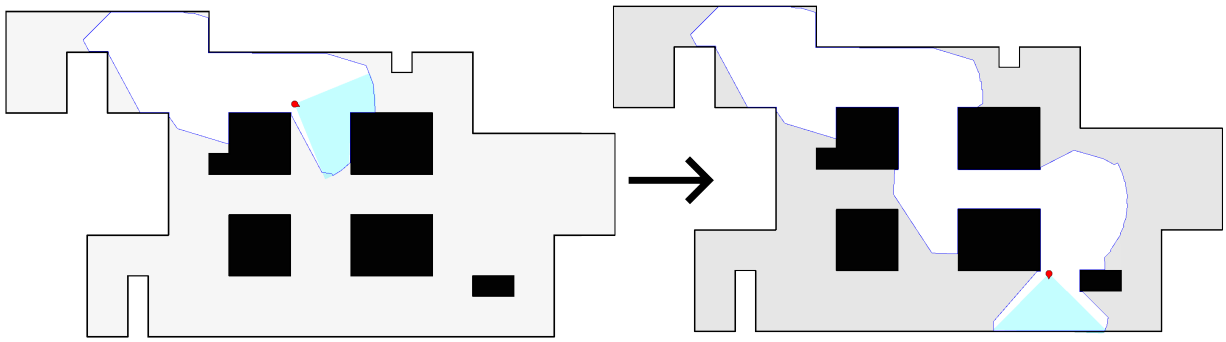


Figure 3.5: The guard moves through the walkable space expanding the covered region with their FOV. Taken from [3]

- Uncovered region: If we geometrically subtract the covered region from the game level region, we obtain the uncovered region. It represents the set of polygons the guards did not cover yet. An illustration of how the covered region grows in a single-guard scenario is depicted in Figure 3.6. The exploration of the game level is considered complete when the covered region equals the game level region.

After that, we partition uncovered regions into a mesh of convex polygons. Any convex decomposition algorithm can be used. For our implementation, we chose Hertel-Mehlhorn [55]. This method will likely produce a reasonable number of convex polygons for computational efficiency. The resulting set of convex polygons within the uncovered region is referred to as uncovered polygons and forms the Visibility Mesh (VisMesh). The convexity requirement for VisMesh polygons simplifies the construction and decision-making process, but is not strictly necessary. More details on that are in section 3.4.2.

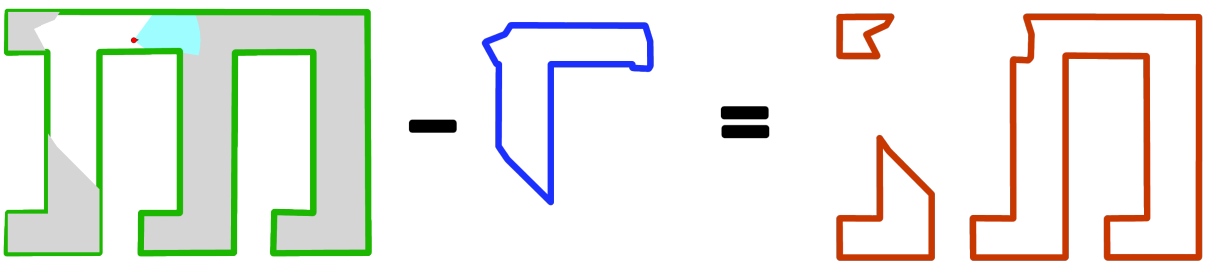


Figure 3.6: The green polygon represents the game level, and the blue polygon represents the growing covered region as the guard moves. The uncovered region is displayed as red polygons, obtained by geometrically subtracting the blue polygon from the green. Taken from [3].

In the VisMesh, every convex polygon is assigned a staleness value, which is a normalized value between 0 and 1, representing the amount of time that has passed since a guard roughly covered the area of the polygon. The guards use this information to guide their patrol motion. Again, this representation must be constantly updated.

3.4.1 Representation-Update

One loop of updating the VisMesh goes as follows:

- At the beginning of the scenario, the covered region is empty. The region is then initialized with the merged FOVs of the guards. In future iterations, the covered region will increment until its area reaches a certain percentage of the game level's area, which we refer to as the coverage threshold. It is reset once the covered region area reaches the coverage threshold. Higher coverage threshold will promote an exploration-like behavior where guards are less likely to revisit areas they covered. Lower values, on the other hand, enable guards to keep revisiting previously covered areas.

- The uncovered region is then calculated by geometrically subtracting the covered region from the game level region. Then, this uncovered region is decomposed into a set of convex polygons, which are the units the guards need to cover. The staleness value

of each polygon indicates the time the corresponding location of the polygon was last covered. At the start of the scenario, the staleness of all uncovered polygons is set to 1. However, in subsequent iterations, the staleness of each polygon is updated in every iteration, based on the staleness of polygons in VisMesh from the previous iteration.

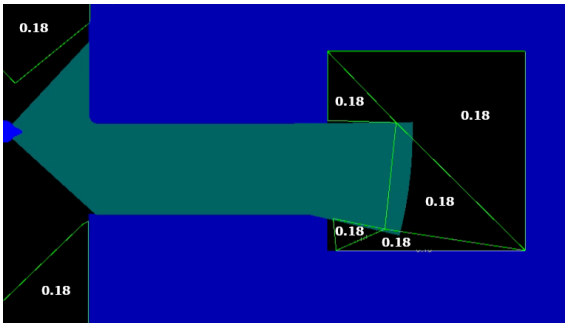
- The decomposition will not generally result in identical polygons due to the constant change in the uncovered region. As a result, we need to find a way to map the staleness associated with the previous VisMesh onto the other. To determine the staleness of VisMesh polygons, we require the VisMesh from the previous loop; figure 3.7 shows a brief explanation of the VisMesh construction. For each polygon in the current VisMesh, we find the intersection areas of this polygon and polygons from the previous VisMesh. Then, there are several subtle ways the staleness can be calculated:

  - Assign the weighted staleness average of the area intersections of the polygons of previous VisMesh that overlap with the polygon.
  - Assign the staleness of the polygon with the highest staleness among the polygons of previous VisMesh overlapping with the polygon.
  - Assign the staleness of the polygon of previous VisMesh that has the largest intersection area with the polygon.

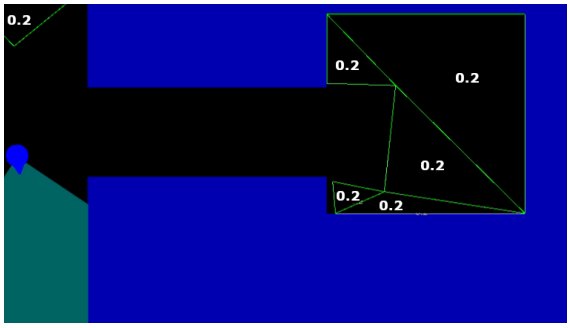
  For our implementation, we opted for the third variation for its simplicity. However, further testing can be conducted to compare these variations as part of future work.

- After updating the VisMesh staleness values, we normalize them to rescale the staleness values of the polygons.

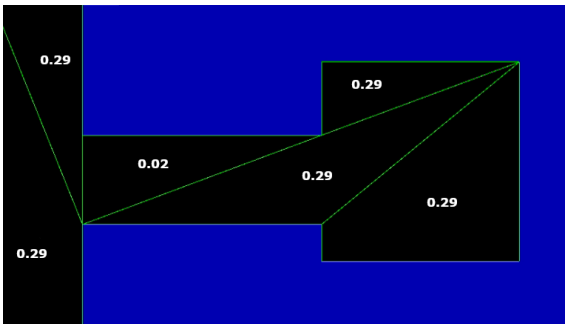
Algorithm 3 summarizes the steps involved in constructing a VisMesh. First, we find the total area of the covered region; if it measures higher than the $COVERAGE\_THRESHOLD$, it is reset (lines 1-3). After that, the covered region is updated by merging it with the guards' FOVs, which gives us the uncovered region when we subtract from the game level



(a) Polygons in the VisMesh outlined in green with the staleness value in the centroid. Once an area is covered, it is removed from the VisMeshs.



(b) As the guard moves, more area is carved from the uncovered region. The covered area is considered to have a staleness of 0.



(c) Once it is time to reset the covered region, we use the staleness of the polygons in the previous VisMesh to calculate the next VisMesh.

Figure 3.7: Screenshots showing how the VisMesh calculation is done based on the previous VisMesh.

polygons (lines 4,5). Following that, we decompose the uncovered region to a set of polygons based on Hertel-Mehlhorn (line 6). After we obtain the set of the uncovered polygons, we calculate their staleness by obtaining the staleness of the polygon with the highest overlap area from the previous VisMesh (lines 7-24). Lastly, we normalize the staleness of the uncovered polygons, which gives us the current VisMesh (line 26).

Reconstructing the VisMesh at every time step is computationally expensive because it requires triangulating the space and decomposing it into convex polygons. To handle that, we update the VisMesh every fixed duration of time t. t should be small enough that guards

Algorithm 3 VisMesh Update

Require: G, the set of guards.
Require: GL, the polygons of the game level.
Require: *covered_region*, the set of polygons that represents the covered region.
Require: $COVERAGE\_THRESHOLD$, the area at which the *covered_region* will be reset.
Require: $V_{previous}$, the previous VisMesh.

if $Area(covered\_region) >= COVERAGE\_THRESHOLD$ **then**
    $covered\_region \leftarrow \{\}$
end if
$covered\_region \leftarrow Merge(covered\_region, G)$
$uncovered\_region \leftarrow Subtract(GL, covered\_region)$
$uncovered\_polygons \leftarrow HertelMelhorn(uncovered\_region)$
for each $p \in uncovered\_polygons$ **do**
    $p.staleness \leftarrow 0$
    $max\_staleness \leftarrow 0$
    $largest\_intersection\_area \leftarrow 0$
    $empty\_area \leftarrow Area(p)$
    for each $p_p \in \mathcal{V}_{previous}$ **do**
        if $DoIntersect(p, p_p)$ **then**
            $intersection \leftarrow intersectArea(p, p_p)$
            $empty\_area \leftarrow empty\_area - intersection$
            if $largest\_intersection\_area < intersection$ **then**
                $largest\_intersection\_area \leftarrow intersection$
                $max\_staleness \leftarrow p_p.staleness$
            end if
        end if
    end for
    if $largest\_intersection\_area > emtpy\_area$ **then**
        $p.staleness \leftarrow max\_staleness$
    end if
end for
$V_{current} = NormalizeStaleness(uncovered\_polygons)$
return $V_{current}$

will likely have an updated representation for making decisions and large enough not to slow the game performance. After testing on multiple machines, we found a time of 0.5 seconds to be an appropriate value to meet both conditions for our test environment.

3.4.2 Decision-Making

When a guard is idle, they iterate among the polygons in VisMesh to find a polygon they want to cover. Once a polygon is chosen, the guard will use the NavMesh to find the shortest path to its centroid. Then, the guard will cover the centroid of the polygon with the center of their FOV. Being convex, it simplifies the task of choosing the centroid to cover. After that, since in the representation-update that area will be added to the covered region, the covered area will be subtracted, and new polygons will be formed as a result.

We determine the fitness value of each polygon and choose the polygon with the highest fitness value as the target polygon for the guard. A weighted average of its properties determines the fitness of a polygon. These properties are:

- Polygon staleness: A value between 0 and 1 indicates how much time has passed since the corresponding area under a polygon is covered.
- Polygon area: The area of the polygon divided by the area of the game level region. It represents the area portion of this polygon relative to the whole walkable area.
- Path distance ($NormalizedDistance(v, g)$): The normalized path distance from the guard's position to the polygon's centroid v. We normalized this distance by dividing it by the game level diameter.
- Closest other guard distance ($DistanceClosestGuard(v, g)$): This is the normalized distance of the closest guard that is not g to the centroid of the polygon v.

These properties are used to determine the fitness of a polygon by using equation 3.4.

$$f(v) = v.staleness * w_{staleness} + GetAreaPortion(v) * w_{area}$$
$$+(1 - NormalizedDistance(v, g)) * w_{distance} + DistanceClosestGuard(v, g) * w_{separation} \tag{3.4}$$

Where w are the weights that determine the importance of a property in finding the fitness of a polygon. In section 3.5, we explore the impact of these weights on patrol performance. Generally, the main parameters that determine affect this behavior are:

- Max percentage of the covered region: This determines the maximum size of the covered region during the patrol. Higher values ensure exploring larger areas of the game level.
- Area weight(w_{area}): This affects the priority of choosing larger polygons over smaller ones. Higher values motivate guards to prioritize polygons with smaller areas.
- Staleness weight ($w_{staleness}$): This affects how the staleness of the polygon is prioritized. Higher values prioritize "staler" areas.
- Distance weight ($w_{distance}$): Higher value prioritizes polygons closer to the guard.
- Separation weight ($w_{separation}$): This weight determines how guards prioritize polygons that are further away from other guards. Higher values encourage guards to separate and cover areas that are more distant from other guards.

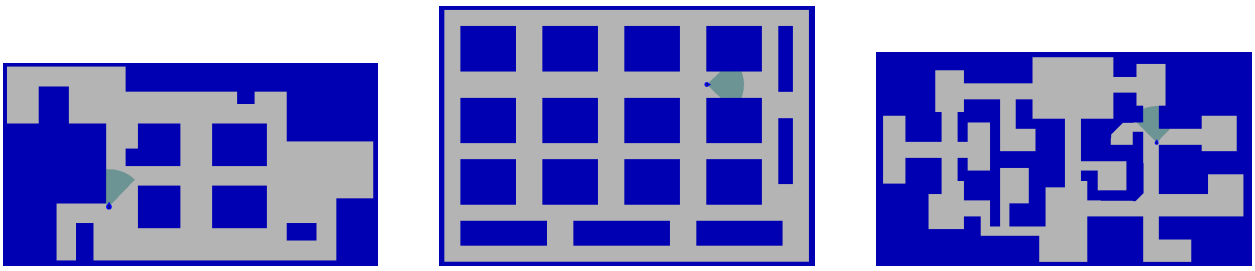
3.5 Weight Tuning

In this section, we evaluate how altering the weights impacts patrol performance. Initially, we will establish the metrics we will employ to evaluate the performance by assessing how effectively the behavior covers the game level on a regular basis. Then, we will investigate the impact of weight modification on these metrics for the three previously defined methods.

3.5.1 Metric

To assess patrol behavior, we establish a criterion of it being "good" if it achieves uniform coverage across the game level for a specified duration. To evaluate this behavior, we employ a discrete data structure. The game level is divided into a grid with W columns and H rows. The dimensions of each grid cell are set to $0.05m \times 0.05m$, ensuring that the values of W and H adapt to the size of the game level. This approach ensures that each cell is sufficiently small to accurately measure coverage. We then measure the duration during which at least one guard covers the center of each grid cell. Finally, we normalize the data for each cell to relatively rescale the covered time for each cell.

We conduct 20 rounds for each parameter setting, and the duration of a round lasts for 120 seconds. A team of four guards with fixed properties, such as FOV radius and angle, movement, and rotation speed, are placed on the game level. All guards follow the same patrol behavior. To analyze the impact on game levels of different properties, we utilize three game levels modeled on maps from commercial games, which are depicted in figure 3.8.



(a) Metal Gear Solid: Dock **(b)** Warehouse **(c)** Among Us : Skeld

Figure 3.8: The game level layouts used in the weights tuning experiments. We designed the warehouse layout as a game level with many junctions; the rest were from commercial games.

3.5.2 Grid-based

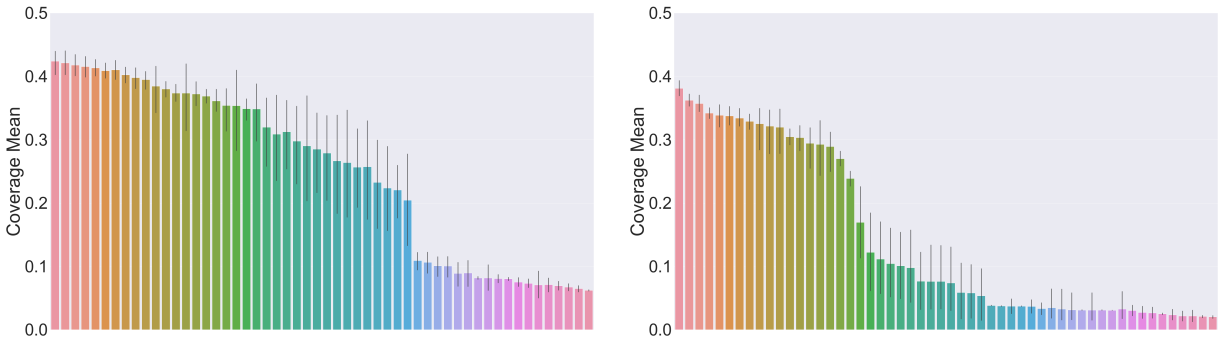
To begin with, we delve into the weight associated with the grid-based representation. We examined various combinations of weights for the parameters discussed in section 3.2.2. The

potential values considered for each weight are listed in table 3.1. Additionally, we compared two different sizes for the cell dimension of the grid: $0.75m \times 0.75m$ and $1m \times 1m$.

Table 3.1: The weight values for the grid-based approach and the possible value we used in our experiments.

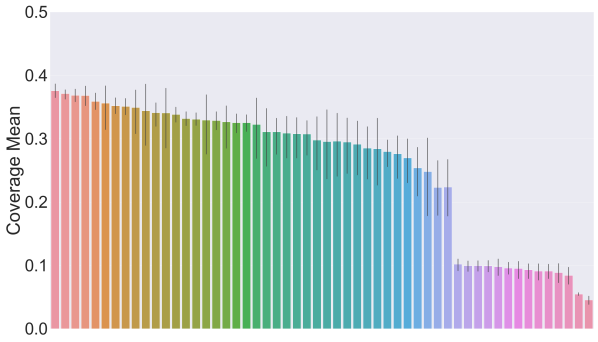
Parameter	**Values**
$CellSize$	$[0.75 \times 0.75,\ 1 \times 1]$
$w_{staleness}$	[0, 0.5, 1]
$w_{distance}$	[0, 0.5, 1]
$w_{separation}$	[0, 0.5, 1]

After each round, we calculate the average of the normalized covered time for cells, which provides an estimate of the game level's uniform coverage. The results of these simulations are presented in figure 3.9; the x-axis represents each combination of the weights; we omitted it for readability. However, for a more detailed analysis, we provide the top 10 weight combinations, sorted by the mean coverage values, in tables 3.2, 3.3, and 3.4, corresponding to each game level.



(a) Metal Gear Solid: Dock

(b) Warehouse



(c) Among Us: Skeld

Figure 3.9: The averages of coverage normalized time (y-axis) for each weight combination (x-axis). We omit the weight-combination labels for readability; see tables 3.2, 3.3, and 3.4 for the top-performing combinations. Each item in the x-axis is made up of the means of 20 120-second rounds run with the corresponding weights. Each map has different properties that potentially affected these results; to compare the map properties, see appendix A.

Table 3.2: The top 10 parameter settings for the grid behavior for the Docks map, ordered by the mean coverage. Each item is coded as follows $CellSize\_w_{staleness}\_w_{distance}\_w_{separation}$.

Rank	Parameters	Coverage Mean
1	0.75_1.0_0.5_0.0	0.42
2	0.75_1.0_0.0_1.0	0.42
3	0.75_1.0_0.0_0.5	0.42
4	0.75_0.5_0.0_0.5	0.41
5	0.75_1.0_1.0_0.0	0.41
6	0.75_0.5_0.5_0.0	0.41
7	0.75_1.0_0.5_1.0	0.41
8	0.75_0.5_0.0_1.0	0.40
9	0.75_1.0_0.5_0.5	0.40
10	0.75_0.5_0.5_0.5	0.39

Table 3.3: The top 10 parameter settings for the grid behavior for the Warehouse map, ordered by the mean coverage. Each item is coded as follows $CellSize\_w_{staleness}\_w_{distance}\_w_{separation}$.

Rank	Parameters	Coverage Mean
1	0.75_0.5_1.0_0.0	0.38
2	0.75_0.5_1.0_0.5	0.36
3	0.75_0.5_0.5_0.0	0.36
4	0.75_1.0_0.5_0.0	0.34
5	0.75_1.0_0.5_1.0	0.34
6	0.75_1.0_0.5_0.5	0.34
7	0.75_0.5_0.5_0.5	0.33
8	0.75_1.0_1.0_1.0	0.33
9	0.75_1.0_1.0_0.5	0.33
10	0.75_1.0_1.0_0.0	0.32

Table 3.4: The top 10 parameter settings for the grid behavior for the Among Us map, ordered by the mean coverage. Each item is coded as follows $CellSize\_w_{staleness}\_w_{distance}\_w_{separation}$.

Rank	Parameters	Coverage Mean
1	0.75_1.0_1.0_0.5	0.37
2	0.75_1.0_0.5_0.5	0.37
3	0.75_1.0_0.5_0.0	0.37
4	0.75_1.0_1.0_1.0	0.37
5	1.0_1.0_0.5_1.0	0.36
6	0.75_0.5_0.5_0.5	0.36
7	0.75_1.0_1.0_0.0	0.35
8	1.0_1.0_0.5_0.5	0.35
9	0.75_0.5_0.5_0.0	0.35
10	0.75_1.0_0.5_1.0	0.34

By studying the tables 3.2, 3.3, and 3.4, we found the following observations:

- A smaller cell size results in better overall coverage. When comparing the top parameters across various maps, a cell size of $0.75 \times 0.75m^2$ demonstrated a higher average coverage compared to $1 \times 1m^2$. This improvement is attributed to the enhanced accuracy achieved when the grid is divided into finer segments. Figure 3.10 displays a bar chart that compares the different cell sizes investigated in our experiments. This outcome was consistently observed across the other maps as well. Additional testing is necessary to determine the optimal cell size at which the average coverage converges.

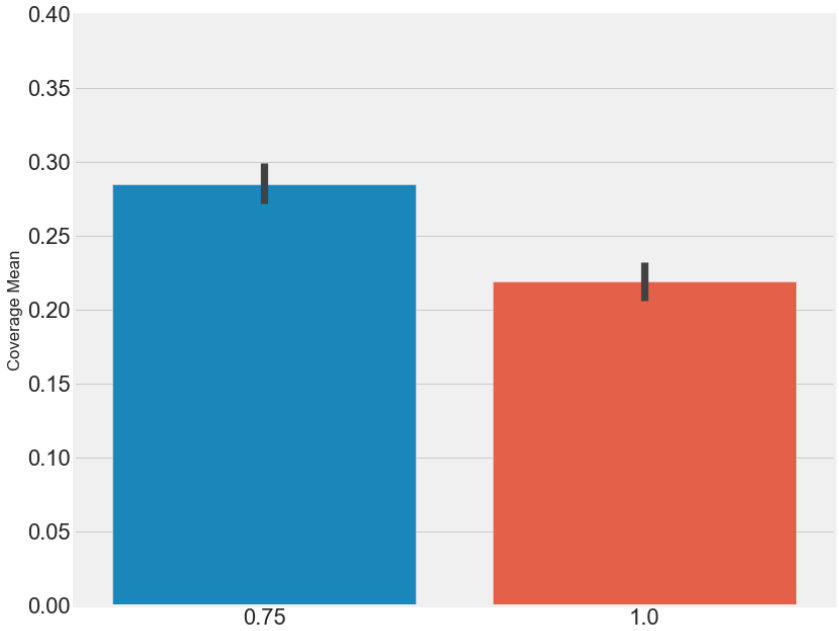


Figure 3.10: The coverage averages normalized time (y-axis) grouped by the two cell sizes we considered in this experiment (x-axis). The results belong to the Metal Gear Solid: Docks map.

- A non-zero value of $w_{staleness}$ led to improved overall performance, assuming $w_{distance}$ and $w_{separation}$ were not zero. The observation is supported by Figure 3.11, which presents a bar chart illustrating the impact of different $w_{staleness}$ values on coverage during patrols.

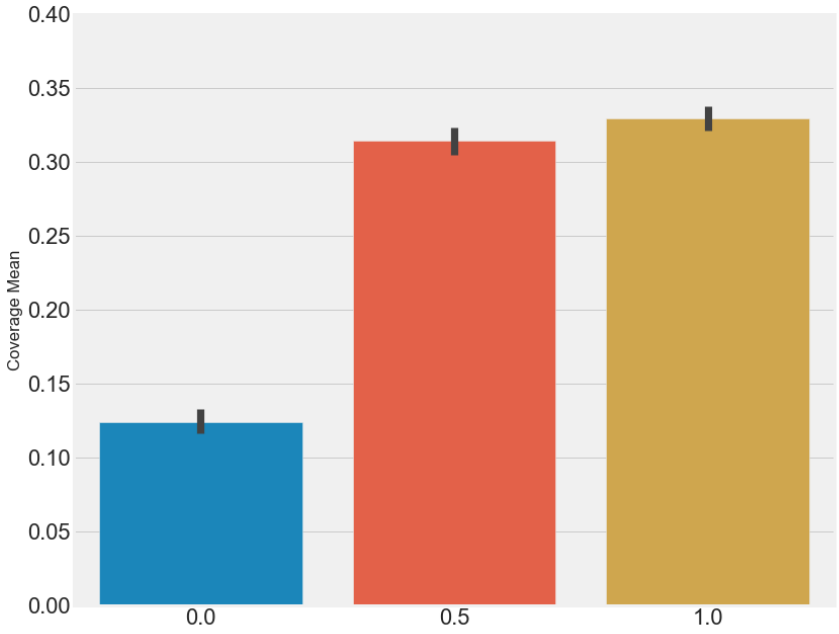


Figure 3.11: The coverage averages normalized time (y-axis) grouped by the $w_{staleness}$ values we considered in this experiment (x-axis). The results belong to the Among Us: Skeld map.

- Upon examining the top parameters, we believe that both $w_{distance}$ and $w_{separation}$ play a role in fostering greater dispersion among the guards, resulting in a more uniform coverage. However, there is no substantial disparity in how each weight affects the behavior relative to the other.

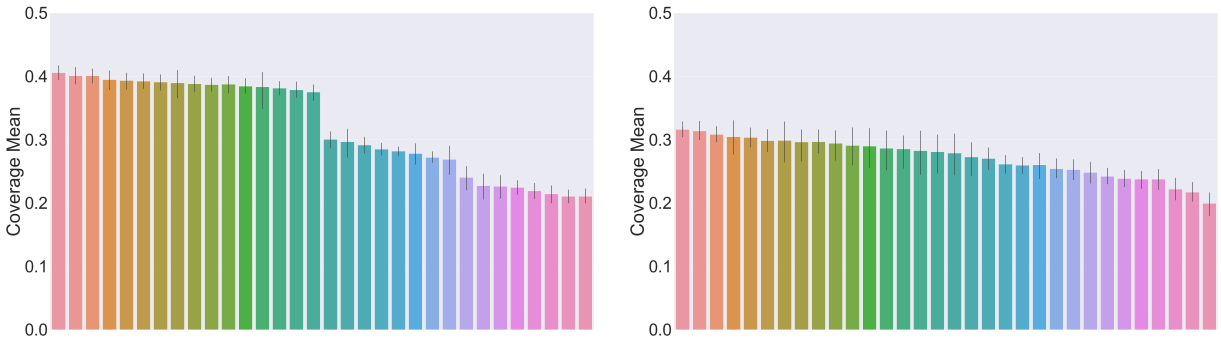
3.5.3 Roadmap

In this section, we explore the impact of the weights on roadmap patrol behavior. As outlined in section 3.3.2. We examine various values for each weight, as shown in table 3.5. Furthermore, we explore parameter configurations for the three maps by conducting 20 rounds, each lasting 120 seconds, for every individual parameter.

Table 3.5: The possible weight values for the Roadmap patrol behavior we consider in the study.

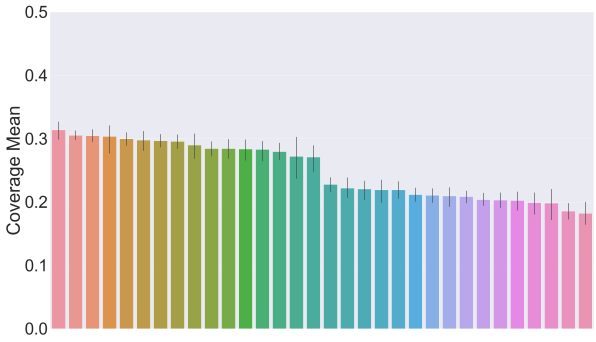
Parameter	Values
Max Path Length	[25%, 100%] of the game level diameter
$w_{staleness}$	[0, 0.5, 1]
$w_{guardPortion}$	[0, 0.5, 1]
$w_{connectivety}$	[0, 0.5, 1]
Passing guards threshold	[Actual, Max]

After running the same number of rounds and settings in the previous method, we also found consistent results between the game levels. Figure 3.12 is a bar chart that shows the performance for weight combinations, and tables 3.6, 3.7, and 3.8 show the top 10 results of these simulations.



(a) Metal Gear Solid: Dock

(b) Warehouse



(c) Among Us: Skeld

Figure 3.12: The averages of coverage normalized time (y-axis) for each weight combination (x-axis). We omit the weight-combination labels for readability; see tables 3.6, 3.7, and 3.8 for the top-performing combinations. Each item in the x-axis is made up of the means of 20 120-second rounds run with the corresponding weights.

Table 3.6: The top 10 parameter settings for the roadmap behavior for the Docks map, ordered by the mean. Each item is coded as follows $MaxPathLength\_w_{staleness}\_w_{guardPortion}\_w_{connectivety}\_PassingGuardsThreshold$.

Rank	Parameters	Coverage Mean
1	1.0_1.0_1.0_0.5_Max	0.41
2	1.0_1.0_1.0_1.0_Max	0.40
3	1.0_1.0_0.5_0.5_Max	0.40
4	1.0_0.5_0.5_1.0_Max	0.39
5	1.0_1.0_0.5_1.0_Actual	0.39
6	1.0_1.0_0.5_0.5_Actual	0.39
7	1.0_1.0_1.0_0.5_Actual	0.39
8	1.0_0.5_1.0_0.5_Max	0.39
9	1.0_1.0_0.5_1.0_Max	0.39
10	1.0_1.0_1.0_1.0_Actual	0.39

Table 3.7: The top 10 parameter settings for the roadmap behavior for the Warehouse map, ordered by the mean. Each item is coded as follows $MaxPathLength\_w_{staleness}\_w_{guardPortion}\_w_{connectivety}\_PassingGuardsThreshold$.

Rank	Parameters	Coverage Mean
1	1.0_1.0_0.5_1.0_Max	0.32
2	1.0_0.5_1.0_0.5_Max	0.31
3	1.0_0.5_0.5_0.5_Actual	0.31
4	1.0_0.5_1.0_1.0_Actual	0.30
5	1.0_1.0_1.0_1.0_Actual	0.30
6	1.0_1.0_1.0_0.5_Actual	0.30
7	1.0_1.0_0.5_1.0_Actual	0.30
8	1.0_0.5_0.5_0.5_Max	0.30
9	0.25_1.0_0.5_0.5_Max	0.30
10	1.0_0.5_0.5_1.0_Max	0.29

Table 3.8: The top 10 parameter settings for the roadmap behavior for the Among Us map, ordered by the mean. Each item is coded as follows $MaxPathLength\_w_{staleness}\_w_{guardPortion}\_w_{connectivety}\_PassingGuardsThreshold$.

Rank	Parameters	Coverage Mean
1	1.0_1.0_0.5_0.5_Max	0.31
2	1.0_0.5_1.0_0.5_Max	0.31
3	1.0_0.5_0.5_0.5_Max	0.30
4	1.0_1.0_1.0_0.5_Max	0.30
5	1.0_1.0_0.5_0.5_Actual	0.30
6	1.0_1.0_1.0_1.0_Max	0.30
7	1.0_0.5_1.0_1.0_Max	0.30
8	1.0_0.5_0.5_0.5_Actual	0.30
9	1.0_1.0_0.5_1.0_Actual	0.29
10	1.0_0.5_1.0_1.0_Actual	0.28

After a closer look at these results, we found the following:

- Elevating the value of "MAX_PATH_LENGTH" led to improved performance. The bar chart supports this observation presented in Figure 3.13, which compares the effect of maximum path length. Similarly, the remaining maps exhibited similar results, further affirming our findings.

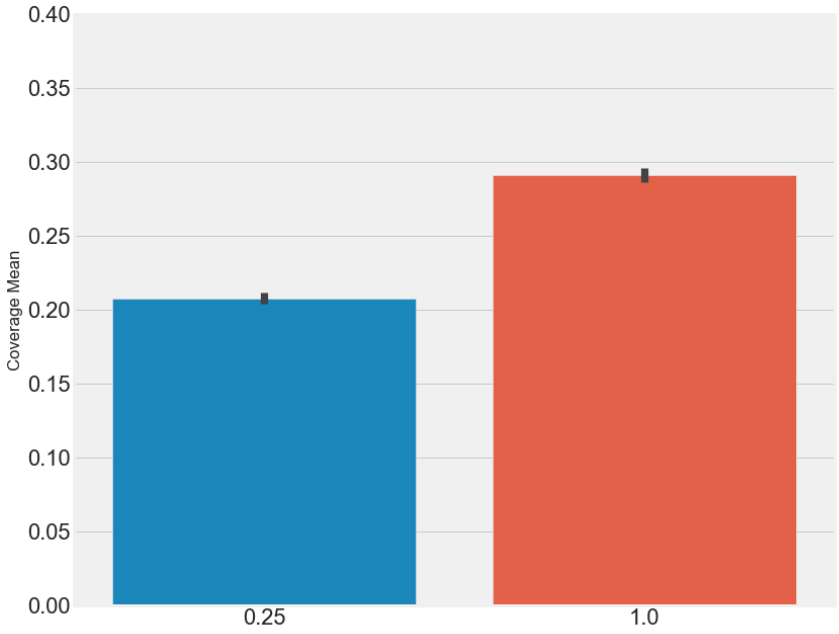


Figure 3.13: The coverage averages normalized time (y-axis) grouped by the two max path lengths, 25%, and 100%, we considered in this experiment (x-axis). The results belong to the Among Us: Skelt map.

- Assigning non-zero weights to $w_{staleness}$, $w_{guardPortion}$, and $w_{connectivity}$ resulted in enhanced performance.
- The "Passing guards threshold" had no significant impact on the overall performance.

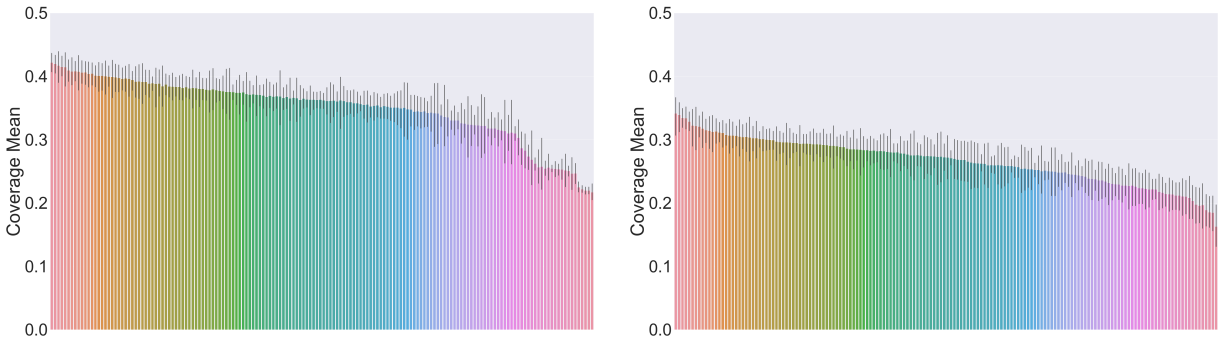
3.5.4 VisMesh

In this section, we examine the impact of weights on VisMesh patrol behavior, as detailed in section 3.4.2. We explore different values for each weight, as outlined in table 3.9.

Table 3.9: The possible weight values for the VisMesh patrol behavior we consider in the study.

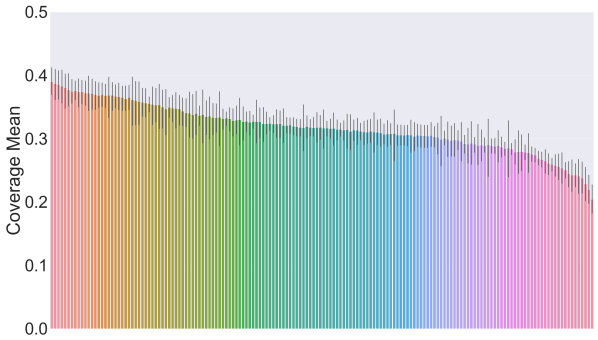
Parameter	Values
Max percentage of the covered region area	[50%, 90%]
$w_{staleness}$	[0, 0.5, 1]
w_{area}	[0, 0.5, 1]
$w_{distance}$	[0, 0.5, 1]
$w_{separation}$	[0, 0.5, 1]

Similar to the previous methods, we ran 20 rounds each lasting 120 seconds, for each combination of the weights and got the coverage mean as an indicator of uniform coverage. Figure 3.14 shows a bar plot of the performance of each weight combination, and tables 3.10, 3.11, and 3.12 show the top 10 combinations for each game level layout.



(a) Metal Gear Solid: Dock

(b) Warehouse



(c) Among Us: Skeld

Figure 3.14: The averages of coverage normalized time (y-axis) for each weight combination (x-axis). We omit the weight-combination labels for readability; see tables 3.10, 3.11, and 3.12 for the top-performing combinations. Each item in the x-axis is made up of the means of 20 120-second rounds run with the corresponding weights.

Table 3.10: The top 10 parameter settings for the VisMesh behavior for the Docks map, ordered by the mean coverage. Each item is coded as follows Max covered region_$w_{staleness}$_w_{area}_$w_{distance}$_$w_{separation}$.

Rank	Parameters	Coverage Mean
1	0.9_0.5_0.5_0.5_0.5	0.42
2	0.9_0.0_0.0_0.5_1.0	0.42
3	0.9_0.5_1.0_0.5_1.0	0.42
4	0.9_1.0_0.0_1.0_1.0	0.41
5	0.9_1.0_1.0_1.0_1.0	0.41
6	0.9_1.0_0.0_0.5_1.0	0.41
7	0.9_1.0_1.0_0.5_0.5	0.41
8	0.9_0.0_0.0_0.0_0.5	0.41
9	0.9_0.5_0.0_0.5_1.0	0.41
10	0.9_1.0_0.5_0.5_0.5	0.41

Table 3.11: The top 10 parameter settings for the VisMesh behavior for the Warehouse map, ordered by the mean coverage. Each item is coded as follows Max covered region_$w_{staleness}$_w_{area}_$w_{distance}$_$w_{separation}$.

Rank	Parameters	Coverage Mean
1	0.9_0.5_0.0_0.5_0.0	0.34
2	0.9_1.0_0.5_1.0_0.0	0.34
3	0.9_1.0_1.0_1.0_0.0	0.33
4	0.9_0.5_0.0_1.0_0.0	0.33
5	0.9_1.0_0.0_1.0_0.0	0.33
6	0.9_0.5_1.0_1.0_0.0	0.32
7	0.9_1.0_0.0_0.5_0.0	0.32
8	0.9_0.0_0.5_0.0_0.5	0.32
9	0.9_0.5_0.5_1.0_0.5	0.32
10	0.9_0.0_0.0_0.0_1.0	0.32

Table 3.12: The top 10 parameter settings for the VisMesh behavior for the Among Us map, ordered by the mean coverage. Each item is coded as follows Max covered region_$w_{staleness}$_w_{area}_$w_{distance}$_$w_{separation}$.

Rank	Parameters	Coverage Mean
1	0.9_1.0_1.0_0.5_0.5	0.39
2	0.9_0.5_0.0_1.0_1.0	0.39
3	0.9_0.5_0.5_0.5_0.5	0.39
4	0.9_0.5_0.5_1.0_1.0	0.38
5	0.9_0.0_0.5_0.5_1.0	0.38
6	0.9_1.0_1.0_1.0_0.5	0.38
7	0.9_0.5_1.0_0.5_0.5	0.38
8	0.9_0.0_1.0_0.5_1.0	0.37
9	0.9_0.0_1.0_0.5_0.5	0.37
10	0.9_0.0_0.0_0.5_1.0	0.37

From these results, we observe the following:

- Enhancing the maximum percentage of the covered region area led to improved uniform coverage. As this value approaches 100%, guards are more inclined to cover the entire game level to reset the coverage. Conversely, with smaller maximum percentages, guards tend to revisit regions they have recently covered. The bar chart supports this observation presented in Figure 3.15, which compares the impact of the maximum coverage percentage on the coverage mean for the "Metal Gear Solid: Dock" map. The patrol performance was notably better when the maximum coverage percentage was set to 90%.

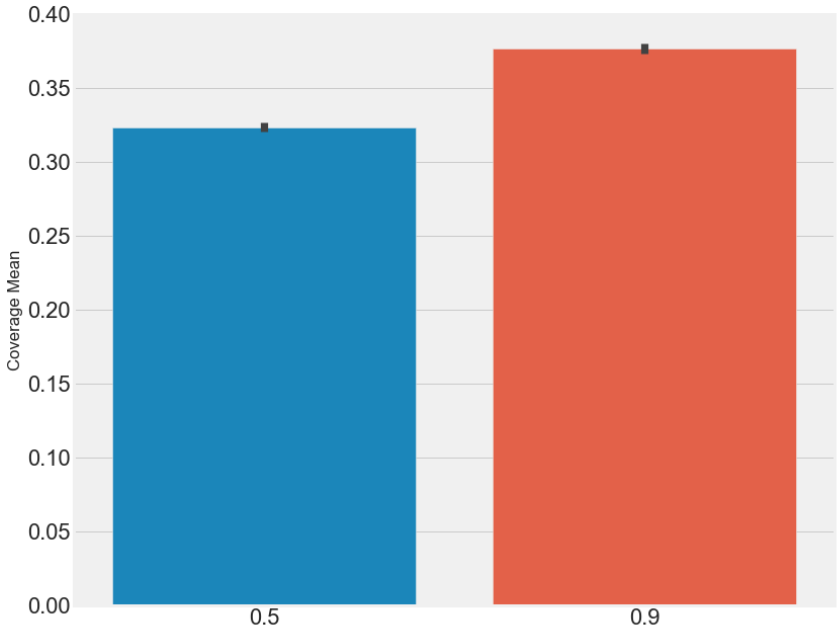


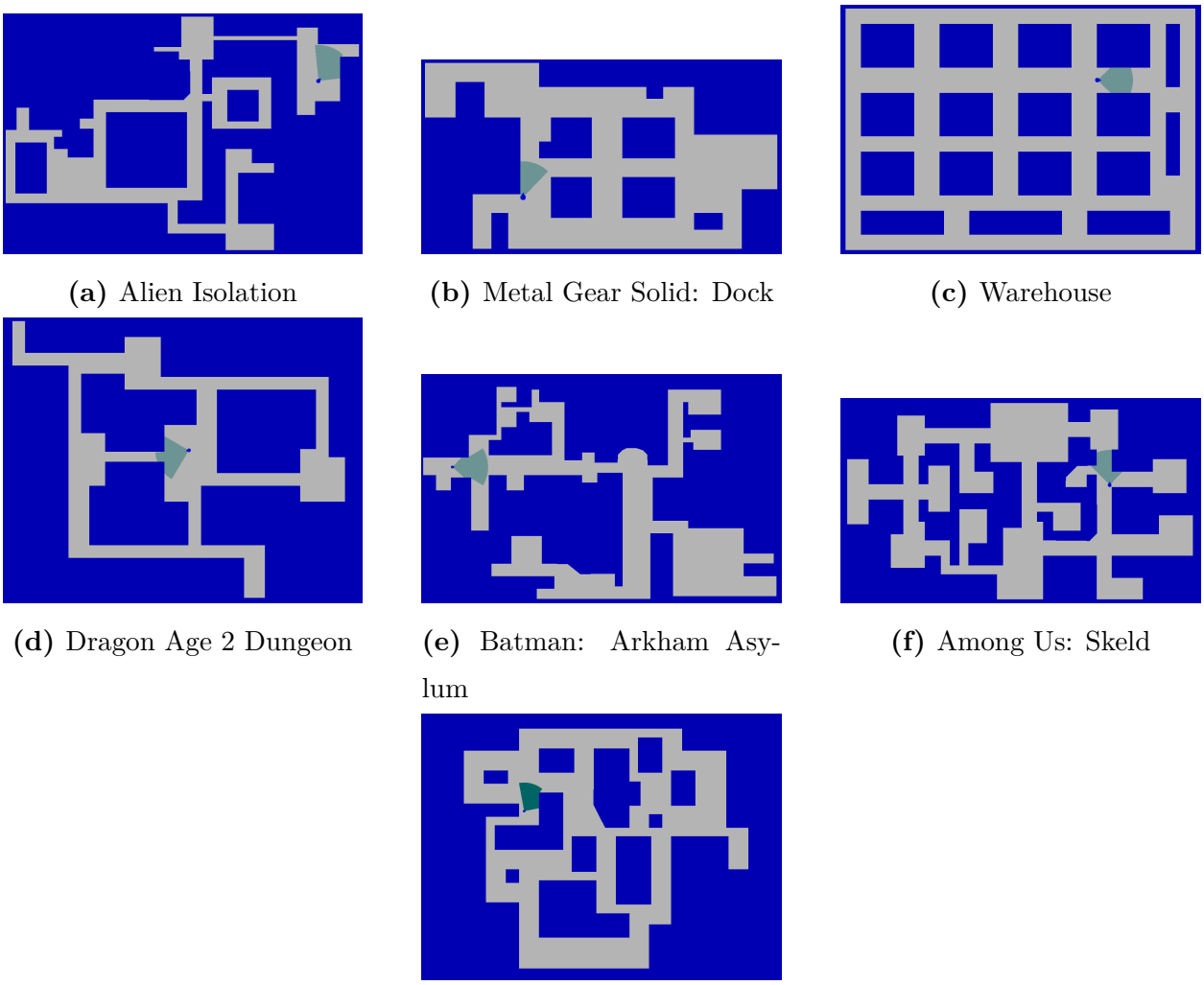
Figure 3.15: The coverage averages normalized time (y-axis) grouped by the max coverage percentage values we considered in this experiment (x-axis). The results belong to the Metal Gear Solid:Dock map.

- The weight value of $w_{staleness}$ did not have a significant impact on patrol performance. This could be because the staleness value for all polygons in the covered region was set to the same value, so the order of coverage for these polygons cannot be determined once the covered region is reset. This could be addressed by keeping track of when certain areas are added to the covered region, so that after a reset, the staleness of polygons will be more accurate instead of having the same staleness value on reset.

3.6 Methods Performance

In this section, we compare the results of the patrol methods based on their game level coverage and computational cost for several setups. We ran several simulations with settings of the following combinations of the following elements:

- Game level maps: We included maps from commercial games and Moving AI 2d pathfinding benchmark maps [106]. Figure 3.16 shows screenshots of the maps we chose in this experiment.



(a) Alien Isolation

(b) Metal Gear Solid: Dock

(c) Warehouse

(d) Dragon Age 2 Dungeon

(e) Batman: Arkham Asylum

(f) Among Us: Skeld

(g) Valorant: Ascent

Figure 3.16: The game level layouts included in this experiment.

- Number of guards: To know how the number of guards will affect the coverage of the game level, we ran a round for each team size from three to six guards.

- Range of FOV: We defined a fixed range that is a percentage of the length of the longest side of the bounding box that covers the game level. We defined it as a percentage to allow the FOV to scale with the size of the map; this also simplifies the need to set

a specific FOVs range for each map. In our experiment, we set the range to be 10% of the length of the longest side of the bounding box of the game level. Exploring the impact of different ranges of FOVs can be done as part of future work.

We consider the following methods; using the parameters determined by the experiments done in the previous section.

- Grid-based: This is the method defined in Section 3.2. Table 3.13 shows the values we choose.

Table 3.13: The values chosen for the grid method.

Weight	**Value**
Cell dimension	$0.75m \times 0.75m$
$w_{staleness}$	1
$w_{distance}$	0.5
$w_{separation}$	0.5

- Roadmap: This is the method defined in Section 3.3. Table 3.14 shows our parameters.

Table 3.14: The values chosen for the roadmap method.

Weight	**Value**
Max Path Length	1
$w_{staleness}$	1
$w_{guardPortion}$	0.5
$w_{connectivity}$	0.5
Passing guards threshold	Max

- VisMesh: This is the method defined in Section 3.4. Table 3.15 shows the corresponding parameters.

- Random: As a baseline for a cheap and simple patrol behavior, we included a random patrol behavior where each guard samples a random position on the road map and finds the shortest path toward it. Once the guard reaches it, it samples a new location, and so on.

Table 3.15: The values chosen for the Vismesh method.

Weight	Value
Max percentage of the covered region area	90%
$w_{staleness}$	1
w_{area}	1
$w_{distance}$	1
$w_{separation}$	1

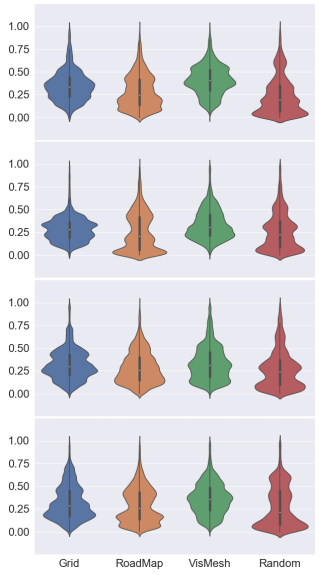
3.6.1 Coverage Performance

We used a heatmap representation to evaluate the coverage performance of the different methods by calculating the total time guards covered each pixel's center. After that, we normalized the total covered time so a pixel of coverage time equals 1 is considered covered the most, and 0 had the least coverage. Figure 3.17 on page 85 shows the violin charts of the coverage performance for different team sizes and methods. Each violin graph is the distribution of the coverage time of the pixels on the map. So if the game level had good coverage, the violin graph would have a larger width on the upper area representing more pixels with a higher coverage value.

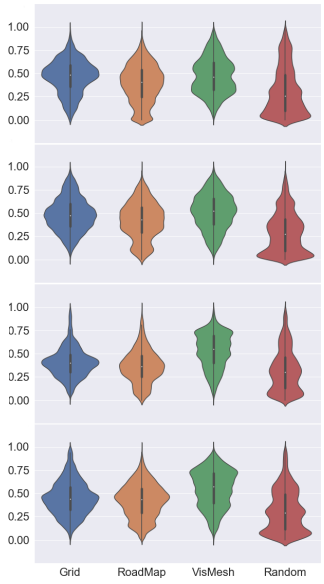
This result shows that the Random method had the worst uniform coverage and did not benefit greatly from adding more guards. This outcome is expected since no model represents which areas are covered and which are not. Additionally, guards had no separation incentive since they independently got their patrol planning. On the other side, the RoadMap method had a better performance; however, we observed on several game level layouts that as the size of the guard team increases, the resulting performance shows more variant coverage values where most pixels of the heatmap are either well covered or the opposite. This could suggest that guards get to cover the areas reachable by the straight skeleton more frequently as the team size increases, but those not reachable by the graph are still neglected. Despite that, the game level layouts with mostly narrow space observable from the straight skeleton benefited more from the increase of guard count, like in the Warehouse and Dragon Age 2 game levels.

Each game level layout possesses distinct characteristics, such as area, diameter, and more. To ascertain whether these properties have a notable influence on patrol performance, we investigated the potential linear correlation between the layout properties and the coverage mean per behavior. However, our analysis revealed no significant correlation between them. For further information on the properties and their definitions, refer to appendix A.

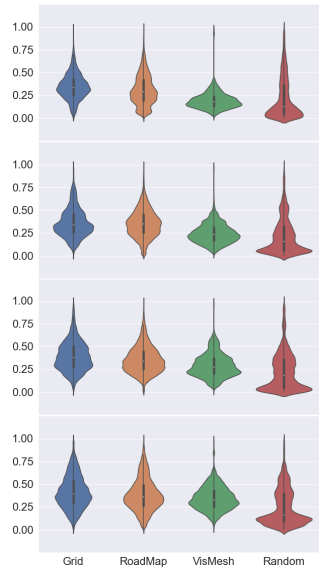
Regarding the most efficient methods, regardless of the guard count, the grid and VisMesh had the best overall coverage across the game level layouts. The heatmaps depicted in figures 3.18, and 3.19 on pages 86 and 87 confirm that these two methods covered more ground than Random or RoadMap. This made the Grid and VisMesh more ideal for extensive patrol tasks. The RoadMap showed decent coverage in the main passageway and corridors of the game level layout, which can provide a lower coverage quality. Finally, Random provides the least coverage, which becomes more evident as the size of the game level layout increases. Furthermore, according to our observations, VisMesh showed an interesting pattern of search-like behavior in covering the nearby regions, while RoadMap behavior guards had a continuous movement along the main paths of the game level. In the next section, we explore the impact of using these methods on the computational cost.



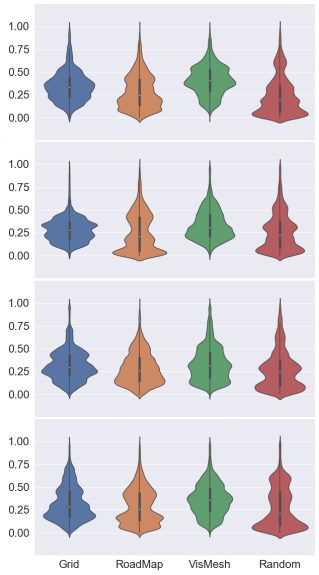
(a) Alien Isolation



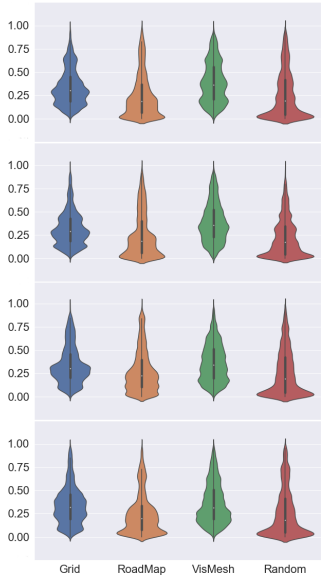
(b) Metal Gear Solid: Dock



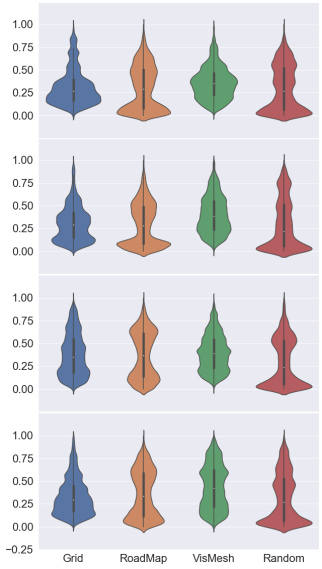
(c) Warehouse



(d) Dragon Age 2 Dungeon



(e) Batman: Arkham Asylum



(f) Among Us: Skeld

Figure 3.17: Violin charts for the patrol performance. Each violin represents the distribution of heat values at the end of a patrol scenario; higher heat values indicate better uniform coverage. Each map represents the performance of a guard team with a specific number, starting with three guards in the first row and the last row for six guards.

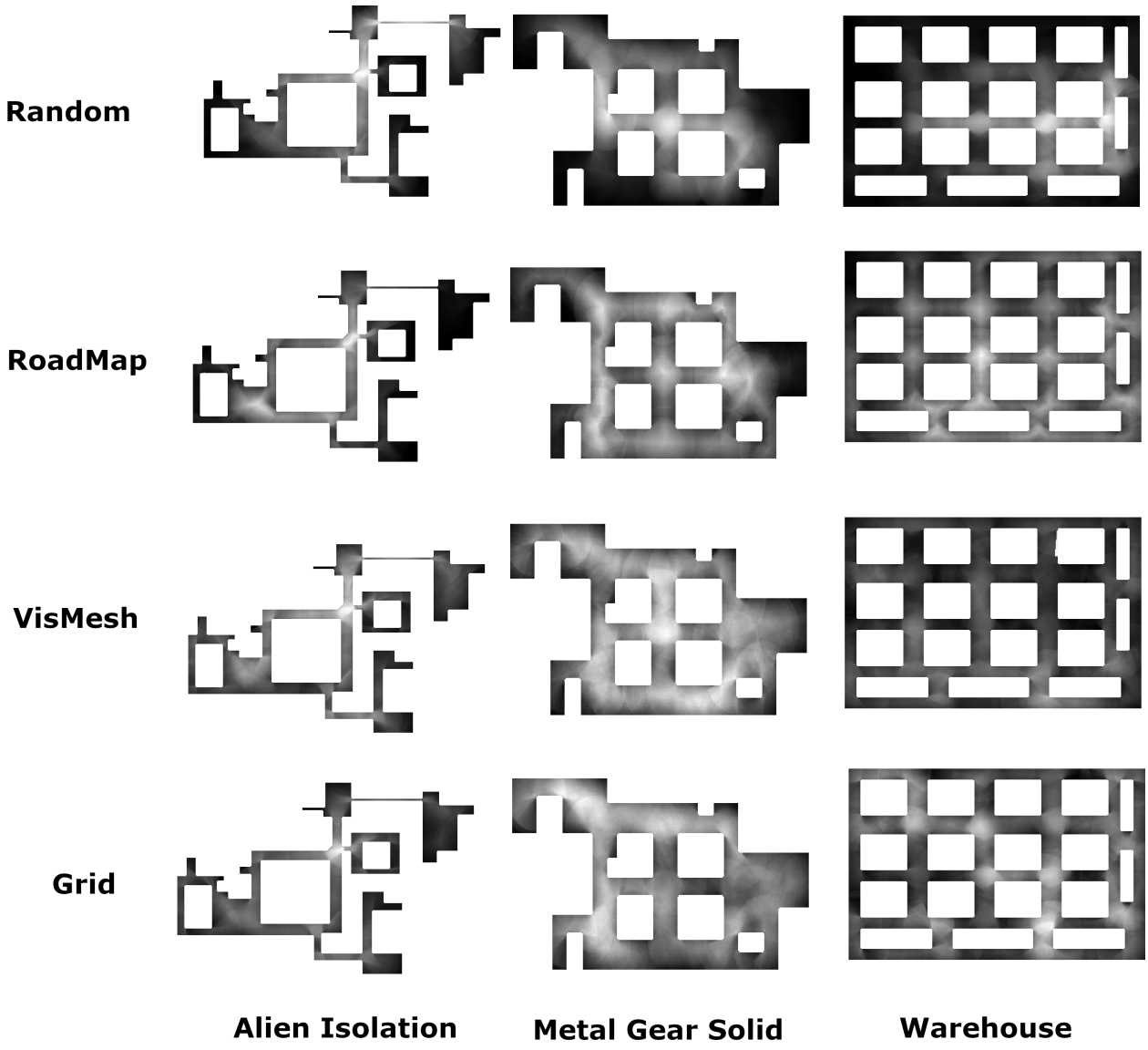


Figure 3.18: The heatmaps for the Alien Isolation, Metal Gear Solid, and Warehouse game levels for the patrol performance of a team of 4 guards. The brighter areas reflect a higher frequency of coverage.

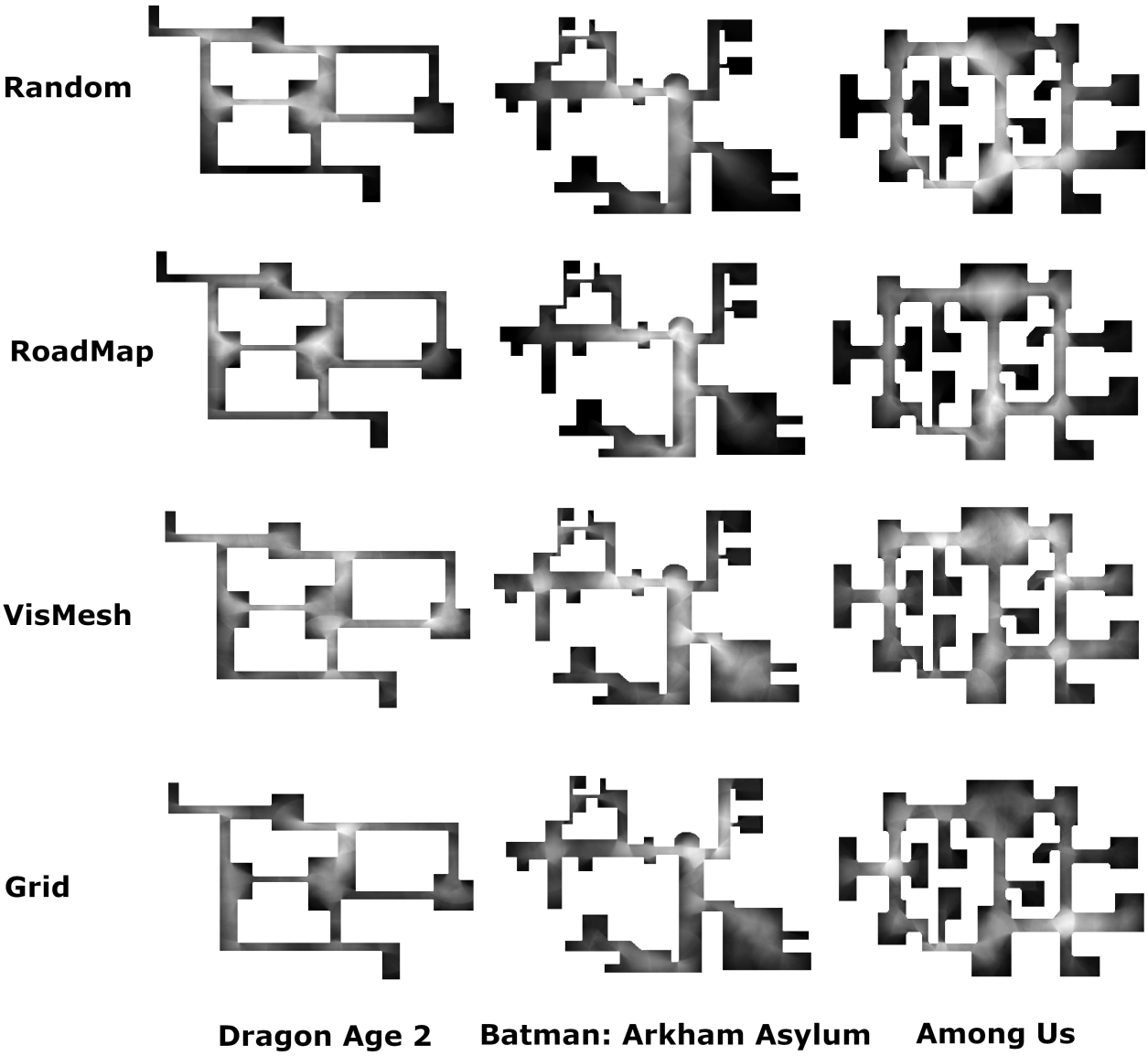
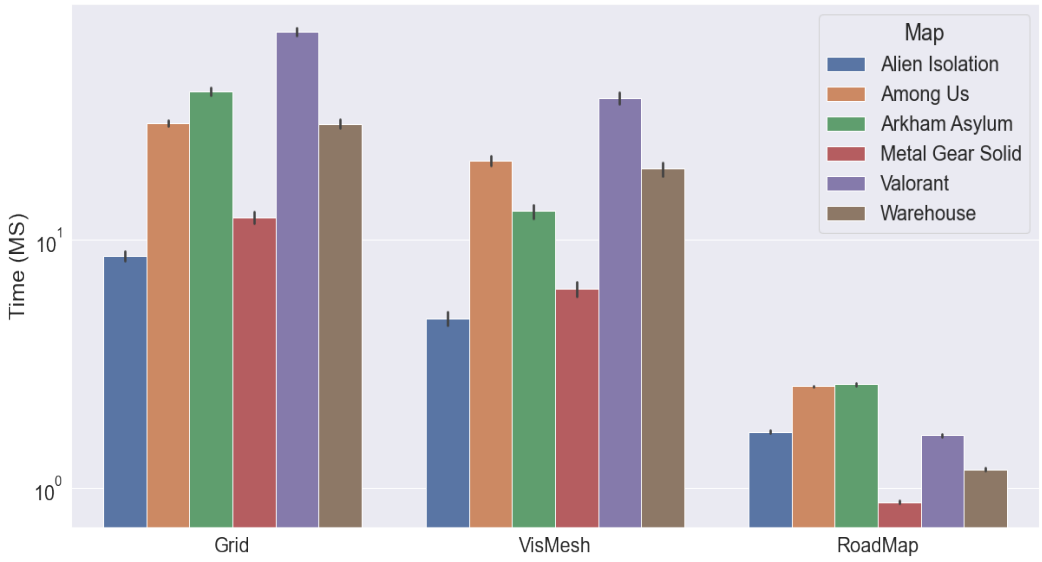


Figure 3.19: The heatmaps for the Dragon Age 2, Batman: Arkham Asylum, and Among Us game levels for the patrol performance of a team of 4 guards. The brighter areas reflect a higher frequency of coverage.

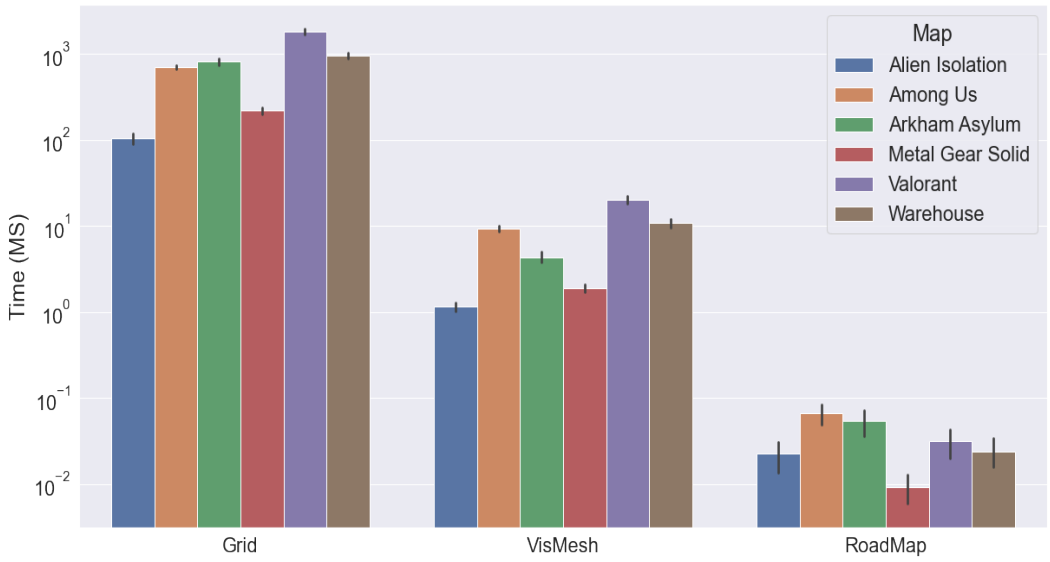
3.6.2 Computational Cost

It is important to assess the extent to which these methods are computationally practical for real-time games. As we mentioned before, each method consists of two main steps: representation update and decision-making. Figure 3.20 on page 89 shows the time the methods take per iteration of these two steps for several maps. In general, we found the

RoadMap to be significantly the cheapest among the three methods, with an average time of 1.7 milliseconds for the update and 0.03 milliseconds for making decisions. This can be due to the sparse representation and simple decision-making technique. After that, the VisMesh proved to be the second most efficient patrol method with an average of 11 milliseconds for the update and 4 milliseconds of the decision-making. The relatively longer update time can be attributed to the recurring decomposition of space. The cost can be further reduced by reducing the frequency of decomposing the space; however, this would have to be balanced to ensure that guards will always have a timely updated VisMesh. Lastly, we found the Grid representation to be the most demanding for most maps. The update step took an average of 28 milliseconds and a much larger decision-making requirement with an average of 667 milliseconds. The high required time for both steps is likely due to maintaining the larger number of cells in the grid. However, we believe that optimizing grid performance can be accomplished through the implementation of more sophisticated data structures, rather than adhering to our current simplistic approach. To reduce the time needed, we can use a coarser grid, which will sacrifice the coverage performance. Additionally, the cell size should be relatively close to the NPC size, so more cells will still be required for larger game levels.



(a) Representation Update



(b) Decision Making

Figure 3.20: The computational performance for the three methods we defined for several maps. The y-axis is the time the method takes in milliseconds; the range is set in log scale. These results were gathered on an Intel® Core™ i5-7500 Processor CPU @ 3.40GHz, 32GB RAM, AMD Radeon R9 200 Series, Windows 10 machine.

3.7 Summary

The objective of this chapter was to develop efficient and effective methods for enabling dynamic guard patrol in stealth games. We introduced three primary variations of the patrol route construction methods, each designed to suit different game level representations: the grid-based, straight skeleton-based, and space-decomposition-based models. We found that each method is capable of generating appropriate patrol behavior with unique features and advantages. However, the VisMesh method produced better patrol results, particularly in consistently covering the game level. Regarding computational efficiency, we observed that the Grid-based representation is unsuitable for real-time applications for larger maps. On the other hand, the other two methods offer significant memory efficiency advantages and are thus more suitable for larger maps. In the next chapter, we investigate how patrol behaviors affect the player experience in more detail.

Chapter 4

Guard Patrol User Study

In the preceding chapter, we presented and assessed several dynamic patrol behaviors that can be used to create guard patrol behavior in stealth games. This chapter outlines the user study we developed to evaluate how these patrol behavior techniques influence the player experience.

Guard patrol behavior is a critical component of a stealth game, as it provides players with a challenge to overcome. Therefore, it is essential to assess how the player perceives this behavior through empirical studies since it is widely acknowledged that it can negatively impact players' experiences, irrespective of its efficiency [73, 124].

We present the contributions of our user study aimed at investigating players' perception of guard patrol behavior in a non-trivial stealth scenario. The scenario requires the player to navigate a space while avoiding detection by guards using specific AI behaviors for patrol. Our previous experiments revealed that the included patrol behaviors have distinct characteristics and efficiency. Thus, we aim to analyze how these traits impact the player experience by assessing self-reported enjoyment and perceived difficulty.

The main contribution of this chapter is a user study to evaluate the effect of different behaviors on player experience. The user study was based on playing with the prototype and survey data. We consider three forms of patrol behavior; two methods are more heuristically complex, and the third is a simple baseline method.

The structure of this chapter is as follows. In Section 4.1, we introduce the scenario that served as a round in the user study. After that, Section 4.2 describes the patrol behaviors that we tested in this study. Then, in Section 4.3, we provide details on the study's setup. After that, in Section 4.4, we present the results of the study, including both player game data and survey responses. Finally, in Section 4.5, we list the possible limitations of this study.

4.1 Study Scenario

Each participant played a game of three rounds. In each round, they competed against guards with unique patrol behavior. To provide players with enough time to observe patterns in guards movement, we set the round to last 120 seconds. In it, the participant controls a character to move around a game level to increase their score by collecting randomly spawned coins. Collecting each coin would increase the player's score by a fixed amount, allowing them to improve their score and incentivize them to navigate to various locations on the game level. The game level is populated with four guards assigned with the same patrol behavior. If the player character is spotted by one of the guards, their score will drop during the period of observation. Note that being observed by guards does not terminate the game; to allow players to focus on the patrol behavior, we programmed the guards to ignore the player's actions and continue their patrol as if nothing had happened. Furthermore, to ensure that the player's actions had an impact at all times in the round, they could have a negative score. Figure 4.1 shows an illustration of the game elements.

We presented the game as a top-down real-time game, with the main elements being:

- Game level: The game level was presented as a polygon with holes. The walkable area is grey, and the unwalkable area is colored with the color we assigned to the current guard team. We framed the game to ensured that all walkable areas were always observable to the player in the game so that they could observe the guards' behavior at all times.

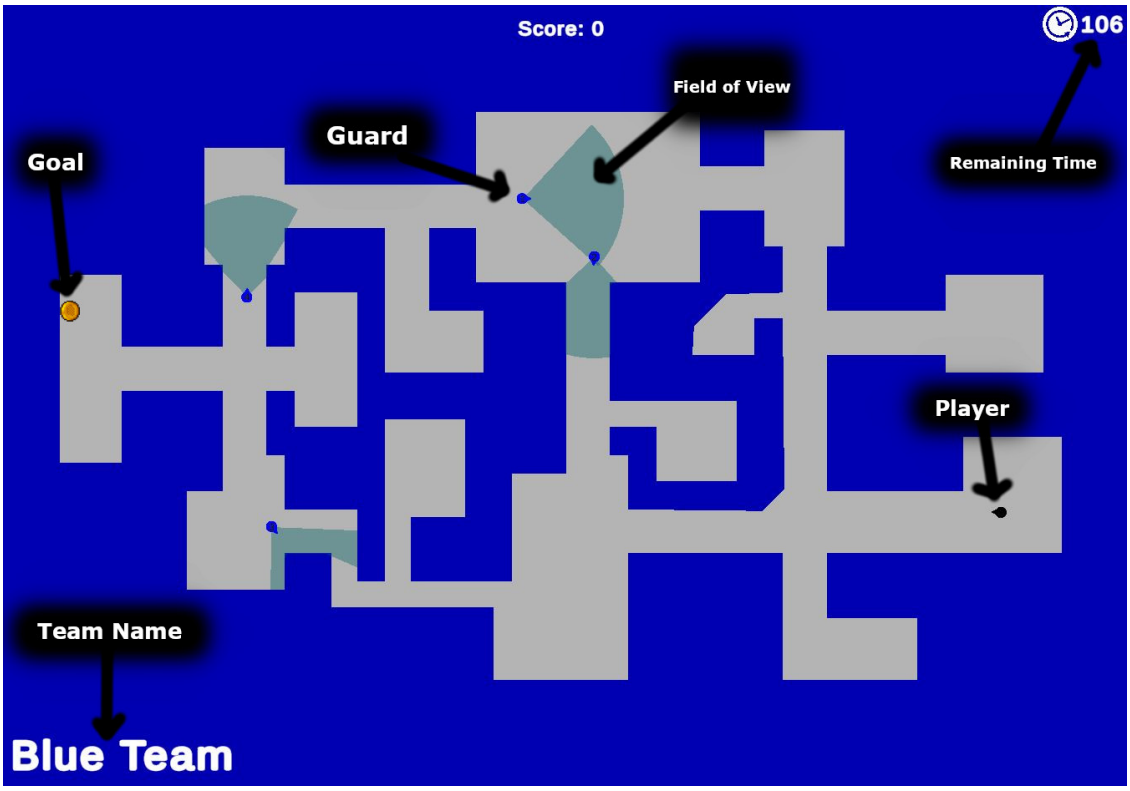


Figure 4.1: A screenshot of the game's features presented to the user study participants.

- Guards: The game level is loaded with four guards randomly positioned. We chose this number of guards to provide players with a moderate challenge and to avoid overwhelming them with many guards to track. The first guard was placed on a randomly selected corner of the outer polygon of the game level. We defined a corner as any vertex on the outer polygon of the game level. Subsequently, the next guard was placed on the corner that is the furthest path distance from the previously placed guards. This process was repeated until all four guards were in place.

- Coin: Like the guards, a coin appears on a random corner at the edge of the game level. The coin serves as the target for the player to increase their score. Once the player picks up the coin, it reappears on the corner that is farthest from their current location. This rule of placement ensures a consistent level of difficulty for all players. During the round, only one coin is available at a time.

- Intruder: The participant controls this character. The player's primary objective is to prevent their score from decreasing by avoiding detection by the guards, and second, to increase their score by collecting coins. To maintain a uniform difficulty level for all participants, the intruder starts at the corner farthest from the coin and guards.
- Time: The end of each round is marked by the passing of 120 seconds, which participants can monitor continuously as the remaining time is displayed on the screen.

Upon completing each round, the player was presented with multiple-choice questions about their perceptions of the guards they had just faced. The order of the rounds was randomly assigned to eliminate any biases. Additionally, each guard team was assigned a random color to assist the player in distinguishing between them. The color was randomized to prevent players from forming associations between the guards' behavior and the color assigned to them. The next section explains the three patrol behaviors we use for each round.

4.2 Patrol Behaviors

Our main objective is to evaluate the player perception of the patrol methods we defined in Chapter 3. In Sections 3.5 and 3.6, we assessed the effectiveness of the guard patrol methods in covering the map to determine the optimal parameter values for these methods. To compare these methods, we divided the map into a grid of WxH nodes and recorded the time each node was within the FOV of a guard during the patrol shift. After the shift, we normalized the coverage time for each node, resulting in values ranging from 0 (the least coverage time) to 1 (the most coverage time). A higher coverage time for each node indicates better patrol behavior, as guards could cover more areas during the patrol shift. The behaviors we include in this study are VisMesh, Roadmap, and Random.

4.2.1 VisMesh

According to our results in Chapter 3, this is the best-performing patrol behavior. The VisMesh method uses triangulation or convex decomposition to partition the area guards cover with their FOVs.

After testing the combinations of parameters for VisMesh, we found the highest average survey time to be 0.49 for the following values: Area weight: 0, staleness weight: 1, distance weight: 1, and separation weight: 0.5. Figure 4.2 shows the heatmap for these parameters.

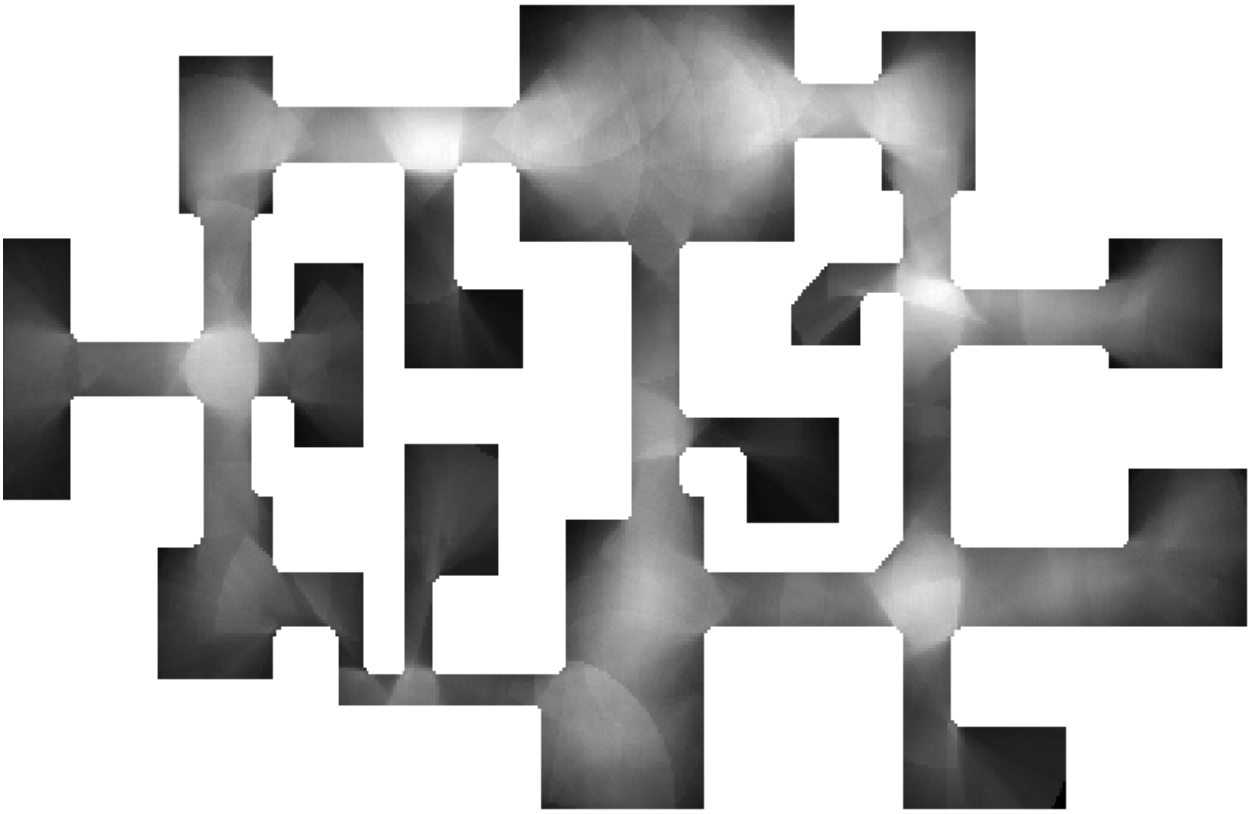


Figure 4.2: Heat maps of a patrol shift where the guards adapted the VisMesh patrol behavior. The brighter a location is, the more coverage it got. We have already shown this heatmap in Section 3.6, but displayed here on a larger scale for easier inspection.

4.2.2 Roadmap

The Roadmap patrol behavior was the second-best method in terms of coverage. It uses the straight skeleton to plan patrol paths for a team of guards.

After preliminary testing, the best parameter combination yielded a staleness average of 0.33 for the following values: Max normalized path length: 0.5, staleness weight: 0.5, passing guard weight: 0, connectivity weight: 0. Figure 4.3 shows the heatmap for these parameters for a sample game level map.

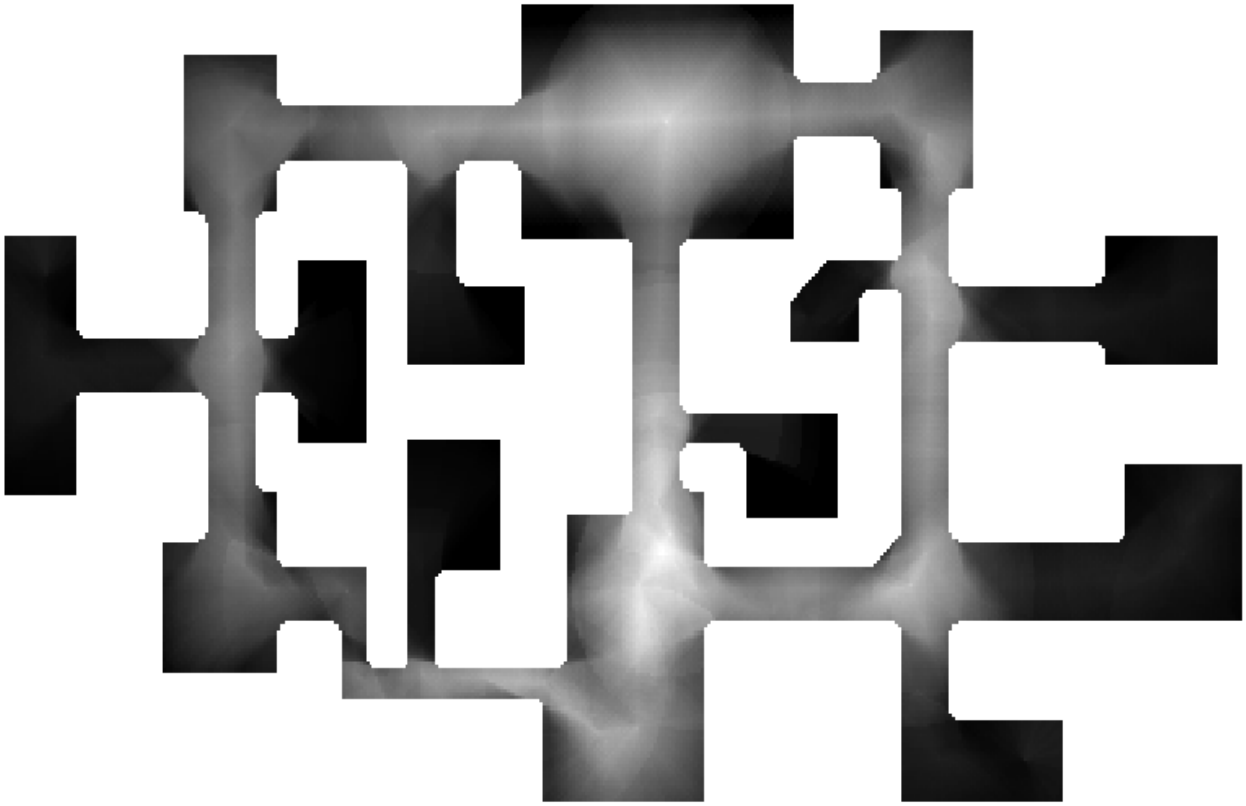


Figure 4.3: Heat maps of a patrol shift where the guards adapted the Roadmap patrol behavior. The brighter a location is, the more coverage it got. We have already shown this heatmap in Section 3.6, we portray it here on a larger scale for easier inspection.

4.2.3 Random

To provide a baseline for comparison, we included a simple patrol behavior. Each guard independently determined the shortest path to a randomly selected position on the level in

this method. Once they reach that position, they randomly choose another position to move to, repeating the process. Figure 4.4 shows the heatmap for this behavior.

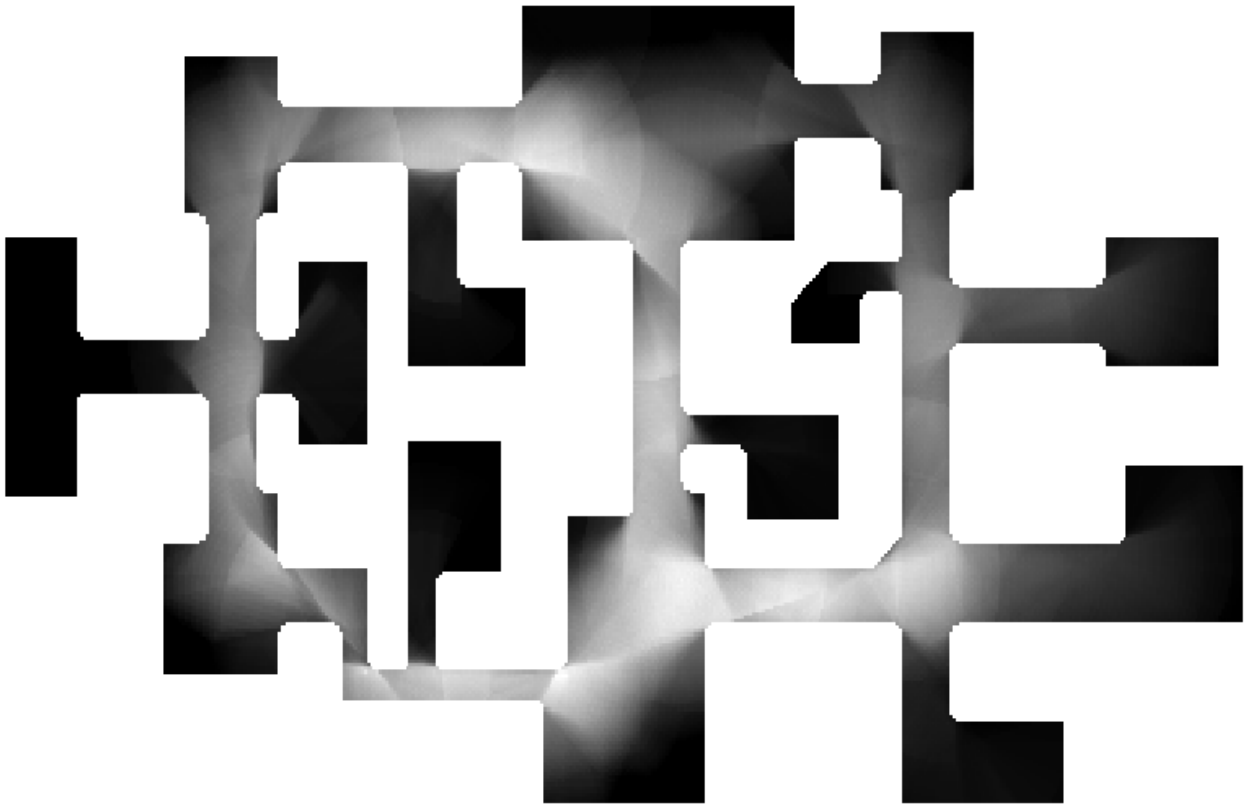


Figure 4.4: Heat maps of a patrol shift where the guards adapted the random patrol behavior. The brighter a location is, the more coverage it got. We have already shown this heatmap in Section 3.6, but shown here on a larger scale for easier inspection.

Prior testing between the three methods showed that Random guards surveyed less ground, while RoadMap and VisMesh had better uniform coverage. Furthermore, it can be seen that the VisMesh had more frequent coverage in the rooms on the map, while the Roadmap had more coverage along the main passages. In the next section, we explain the user study setup.

4.3 Experiment

To gather participants for our study, we utilized email correspondence to reach out to both Undergraduate and Graduate students at McGill University. Our email included a comprehensive overview of the study and a link to an online portal hosting a web-based version of our game. We collected our participants' gameplay data and survey responses anonymously, and they had continuous access to the game through the web portal. The study was completed in an average of 15 minutes.

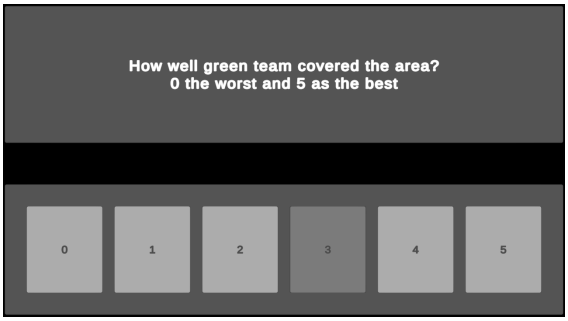
At the beginning of the game, each participant was introduced to the study's aim and instructions on how to play the game. Afterward, they could play a tutorial level to become acquainted with the game's mechanics. Once the tutorial level had concluded, participants could replay the tutorial or begin the actual game if ready, yet our results showed that players rarely chose to replay the tutorial round.

During the main game, participants engaged in three rounds, each featuring a different patrol behavior assigned to the guards. Each team was designated a unique color to assist players in distinguishing between the guard teams. The pairing and order of the guard team and color were randomized between study sessions to prevent potential biases. To encourage participants to reflect on the guards' behavior after each round, we asked them to evaluate three characteristics of the guard's behavior - enjoyability, difficulty, and efficiency. They could rate each aspect on a Likert scale ranging from 0 to 5 to provide a means of reflection. Figure 4.5 shows screenshots of the end-of-round questions.

After finishing all three rounds, participants were asked to specify the teams they found to be the most enjoyable, challenging, and effective. Moreover, we allowed them to explain their selections for each aspect via text input, as illustrated in Figure 4.6. The study concluded at this point, and participants were free to replay the game.

(a) Enjoyment

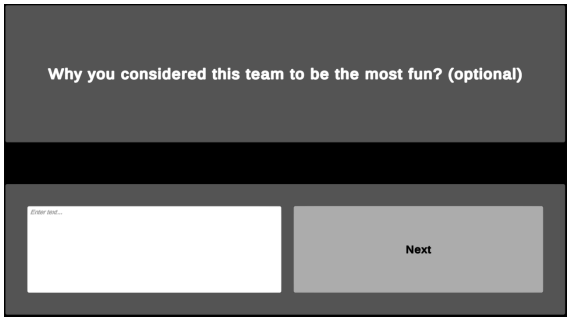
(b) Difficulty



(c) Efficiency

Figure 4.5: The questions asked after the end of each round.

(a) Most enjoyable team



(b) Justification

Figure 4.6: Screenshots of the two questions in the game that pertain to identifying the most entertaining team. We followed them with a similar inquiry about difficulty and effectiveness.

4.3.1 Game Level

In the tutorial round, we employed a Metal Gear Solid map shown in figure 4.7. We chose it for its relative simplicity to familiarize players with the game mechanics. In the primary game, considering the anticipated limited sample size for this user study, we used a fixed

single map to enhance the possibility of obtaining statistically significant results. The map was intentionally designed to resemble the Skeld map from the game "Among Us" [61], offering a moderate challenge with multiple cycles and various enclosed spaces to facilitate player hiding. Figure 4.1 presents the map layout.

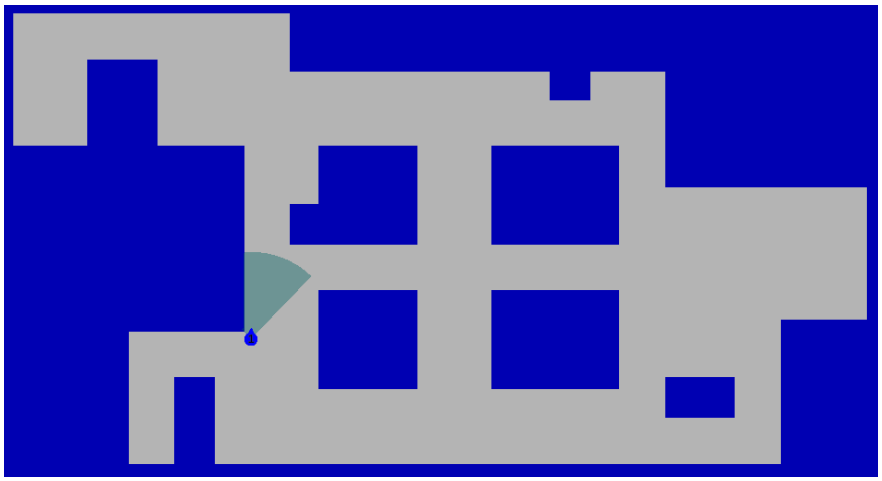


Figure 4.7: The Docks map from the commercial game Metal Gear Solid [68]

4.3.2 Guard Teams

In the tutorial round, we kept the guard count to two to help participants grasp the movement of guards within the game space and avoid overcrowding due to the relatively smaller game level. In the primary game, we tested the guard team formations for several rounds and decided to populate the game level with four guards. As for the assigned behaviors, we assigned the tutorial team with the Random patrol behavior and the following three rounds with the three behavior patterns we described in section 4.2 in random order. In this section, we describe the results of this study.

4.4 Results

In this section, we describe the participation distribution in our study. Following this, we present an evaluation of participant performance based on their gaming background and in

relation to patrol behaviors. Subsequently, we analyze and present the potential factors that may influence player enjoyment and perception of difficulty concerning patrol behavior.

4.4.1 Participation

A total of 115 participants who completed the game were recruited for this study. As part of our data collection, we asked the players to assess their self-perceived familiarity with video games. The distribution of participants based on their experience with video games is illustrated in the bar chart presented in Figure 4.8. Most participants identified themselves as having Advanced or Intermediate experience with games. This distribution indicates that our sample consists of individuals sufficiently familiar with video games, enabling them to provide valuable insights.

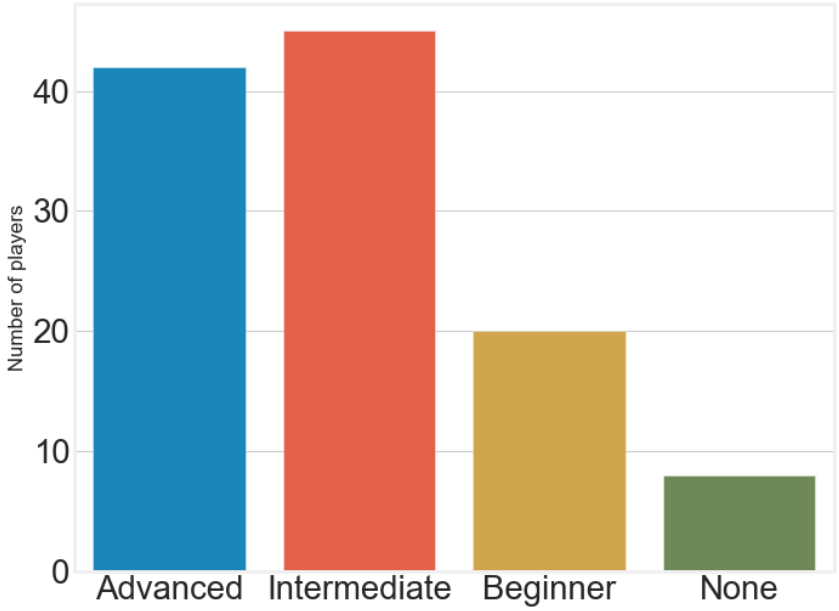


Figure 4.8: The numbers of players who participated in the study grouped by their respective experience in video games. [6]

To verify whether participants of varying experience levels found our prototype reasonably easy to play, we conducted a comparison of the average scores attained by each group

across the three rounds of play. Figure 4.9 shows a bar chart of participants' scores grouped by their experience of video games. It shows that advanced and intermediate players had consistent scores throughout the rounds. Additionally, these participants had a slight increase in scores in later rounds, which could indicate that they became familiar with the game as the rounds progressed. However, for the lower two experience levels, there was a high variance and a small sample size that made it hard to determine a pattern.

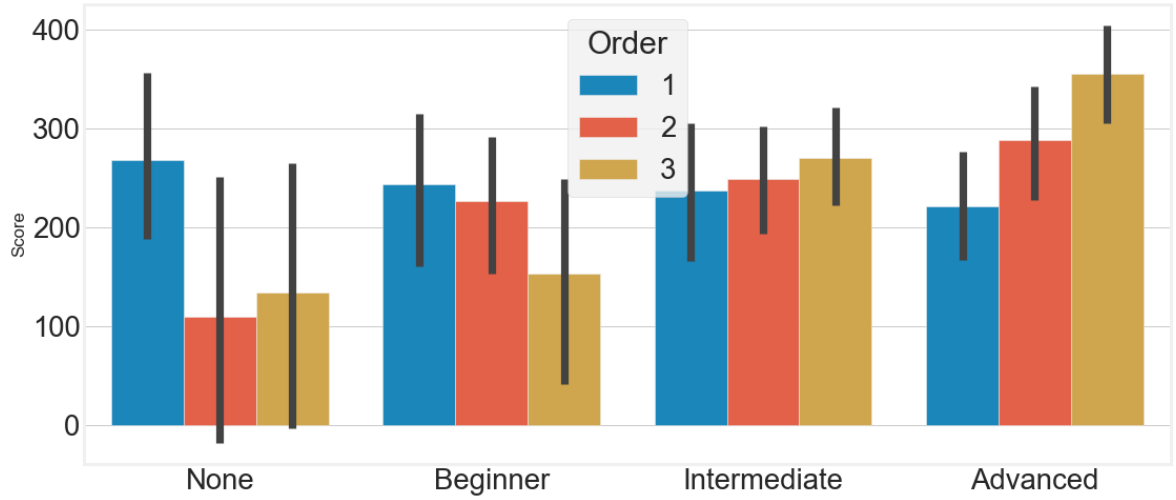


Figure 4.9: The scores participants achieved in the study. The participants sorted themselves into one of four experience levels in video games. Each color represents the round the score was achieved in. The error bars represent 95% confidence intervals.

4.4.2 Performance

To assess whether different guard teams had varying impacts on participant performance, we conducted a comparison of their scores with respect to the round and the type of behavior they faced. Figure 4.10 illustrates a bar chart depicting player scores against each guard team, categorized by the round in which they encountered that particular team. Participants achieved the highest scores when facing the RoadMap team, followed by the Random team, and notably, the VisMesh team yielded the lowest scores. Regarding the influence of the round order on the scores, participants appeared to perform slightly better when encounter-

ing the Random team in later rounds. This might be attributed to players becoming more accustomed to the game with continued play. However, this same trend was not observed with the Roadmap and Vismesh teams. RoadMap seemed to offer a level of ease that allowed consistent scoring regardless of the round order, whereas Vismesh consistently posed a greater challenge.

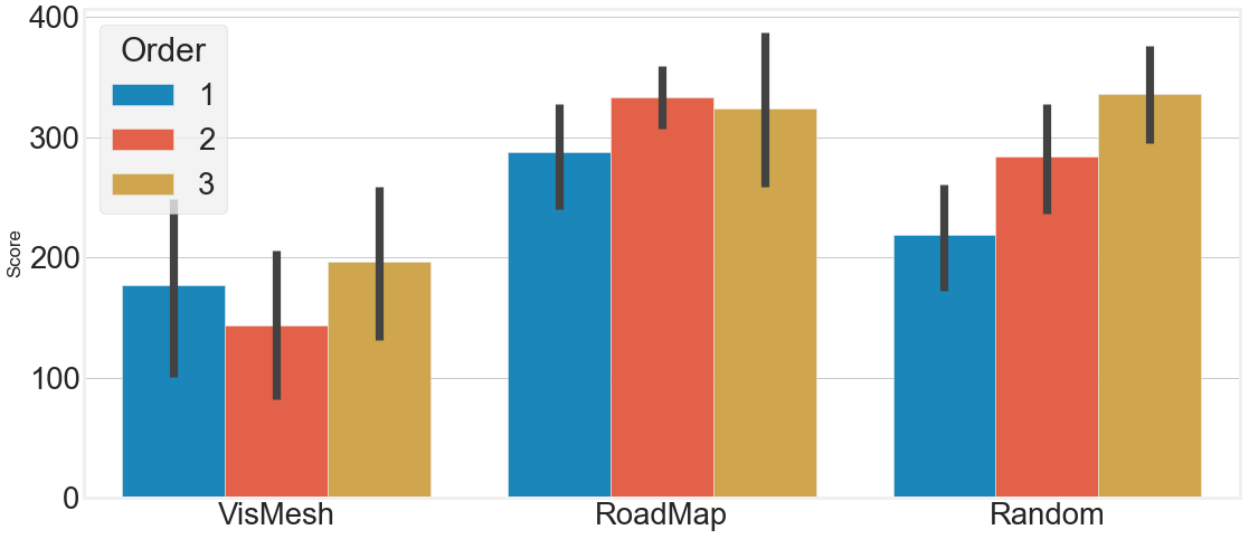


Figure 4.10: Players' scores against the patrol method grouped by the round where they encountered each behavior. The error bars represent 95% confidence intervals. [6]

In the next section, we describe participants' reported enjoyment and analyze the factors that could have affected it.

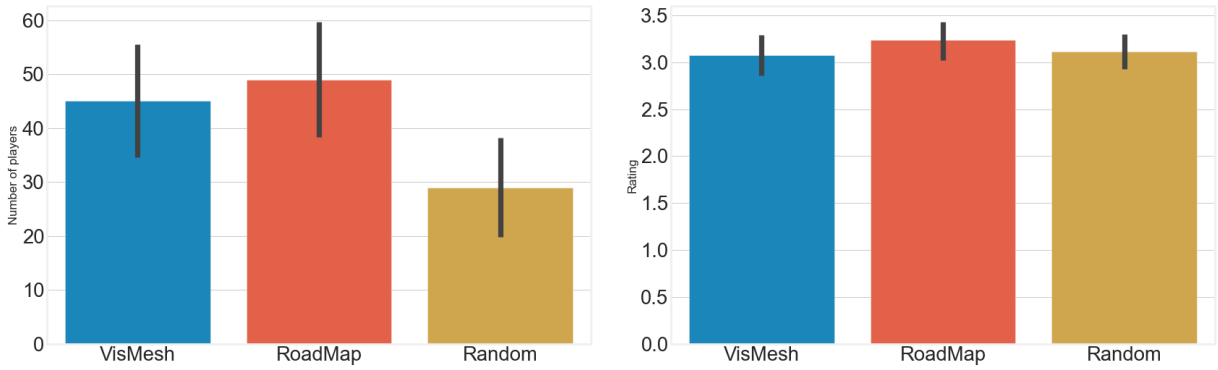
4.4.3 Enjoyment

After finishing each round, we asked participants to rate their enjoyment on a Likert scale from 0 to 5. After that, when they played against all teams, we asked them to choose the most enjoyable team and the option to insert a justification for their choice in free text. Figure 4.11 shows bar charts of player rating of teams in terms of enjoyment. We observe that players had no definite opinion of the enjoyability of a team after immediately playing against them; however, when they competed against all teams, they found the Roadmap and

	χ^2	p
Enjoyment	5.46	0.06
Difficulty	55.76	<0.001

Table 4.1: The Chi-square goodness-of-fit test results of the players' most enjoyable and difficult behaviors. For $\alpha = 0.05$ and degrees of freedom = 2, the critical value is approximately 5.991.

the Vismesh to be the most enjoyable, and fewer players chose random behavior. The End-of-study result could be more reliable because the participants had the chance to compare all teams and hence have a more concrete opinion of the teams. However, a Chi-square goodness-of-fit test showed that a specific behavior was more preferred over other behaviors, as shown in table 4.1.



(a) End-of-study [6] **(b)** End-of-round

Figure 4.11: Players' rating of fun for different teams. The error bars represent a 95% confidence interval.

To understand how participants enjoy playing against a specific behavior, we investigate several factors that might have affected this aspect of the player's experience. First, a Chi-square goodness-of-fit showed no significant impact of the team's color on player enjoyment. Therefore, the other possible factors include:

Order

The sequence in which participants encounter specific behaviors can influence their enjoyment of the team. For instance, playing against a difficult team followed by an easier one may result in a different level of enjoyment compared to the reverse order. To ensure fairness, we verified that the teams were assigned random orders for the study, and no particular ordering was significantly favored over others.

In Figure 4.12, we present a bar chart indicating the patrol behavior participants found most enjoyable, categorized by order of appearance. Generally, players tended to perceive the last team they competed against as the most enjoyable. However, the RoadMap behavior seemed more enjoyable from the second round onwards, suggesting that participants remembered it better. As for VisMesh, many players found it enjoyable regardless of achieving lower average scores against it, indicating an overall preference for VisMesh over Random behavior.

Players may be more inclined to select the last team they played against as the most enjoyable, possibly due to the recency effect [98]. Additionally, player enjoyment could have been influenced by their performance scores. For instance, participants appeared to find RoadMap behavior more enjoyable starting from the second round, potentially indicating a relationship between higher scores and enjoyment.

Surprisingly, even though players generally achieved low scores against VisMesh, many still chose it as their most enjoyable team. This suggests that factors beyond performance scores might have influenced their enjoyment of these behaviors.

Despite these interesting observations, the data collected did not reveal any statistically significant conclusions. Nevertheless, we believe conducting further tests with a larger sample size could increase confidence in the results and provide more insights into the factors impacting players' enjoyment.

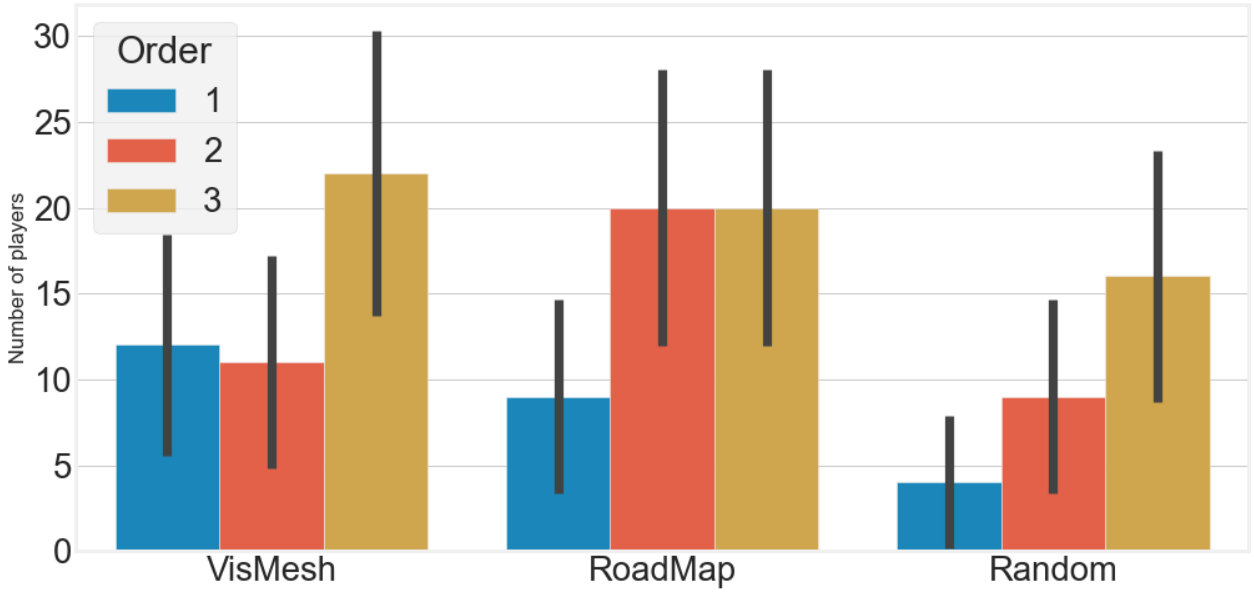


Figure 4.12: The number of players rating patrol behaviors as most enjoyable is grouped by the order in which the behavior appeared. The error bars represent a 95% confidence interval. [6]

Challenge

Upon reviewing the participants' textual justifications for their choice of the most enjoyable team, we discovered that the primary criterion for many players was the level of challenge presented by the guard team. Preferences among players varied, with some favoring easy guards that allowed them to achieve higher scores. In contrast, others were drawn to the most challenging guards because they found them motivating and saw them as an opportunity to improve their gameplay.

Furthermore, we investigated whether players tended to select the team that enabled them to score the highest. Our findings indicated that approximately 36% of the players indeed chose the behavior that allowed them to achieve the best scores. Interestingly, some players enjoyed the behavior simply because it was easy to tackle. However,80% of the players justified their choices and 40% of them preferred the more challenging behaviors.

This observation leads us to believe that there are two main types of players: those who prioritize easier gameplay for higher scores and those who derive enjoyment from the thrill and motivation of facing challenging opponents.

Predictability

A number of players found less predictable guards to be a more enjoyable challenge. It appeared that players derived satisfaction from encountering guards who exhibited more spontaneous behaviors, as these behaviors indicated a certain degree of adaptability and effectiveness. For instance, 20% of the players who provided justifications for their choices expressed a preference for teams they considered unpredictable, with the majority of them singling out the VisMesh guards as embodying this quality.

Effectiveness

A conisderable percentage of players tended to derive more enjoyment from facing a team that appeared adept at their designated task. This inclination might be attributed to players feeling a greater sense of accomplishment when competing against a competent AI adversary. As a part of this study, we requested players to assess effectiveness, allowing us to examine whether players selected the same team as both effective and enjoyable. Our findings revealed that 37% of players regarded teams they deemed enjoyable as also effective. Furthermore, 20% of players characterized the team as well-distributed, while 5% described it as meticulous or natural. Some comments from participants included: “It checked corners of the room it entered, so if it trapped you, you suffered and couldn’t just escape unnoticed by hiding in a corner”, “This team provides a significant challenge by covering a large amount of ground and move in a more natural pattern, scoping the area more thoroughly with movements that seem instinctive.”

4.4.4 Difficulty

Regarding the level of difficulty, participant responses at the end of each round did not reveal any significant differences in perceived difficulty among the teams. However, in the end-of-study responses, players unmistakably singled out a particular team as the most challenging. Figure 4.13 illustrates the results of their responses, showing that participants predominantly selected the VisMesh team as the most challenging, with an equal number of players divided between choosing Roadmap and Random as the most difficult.

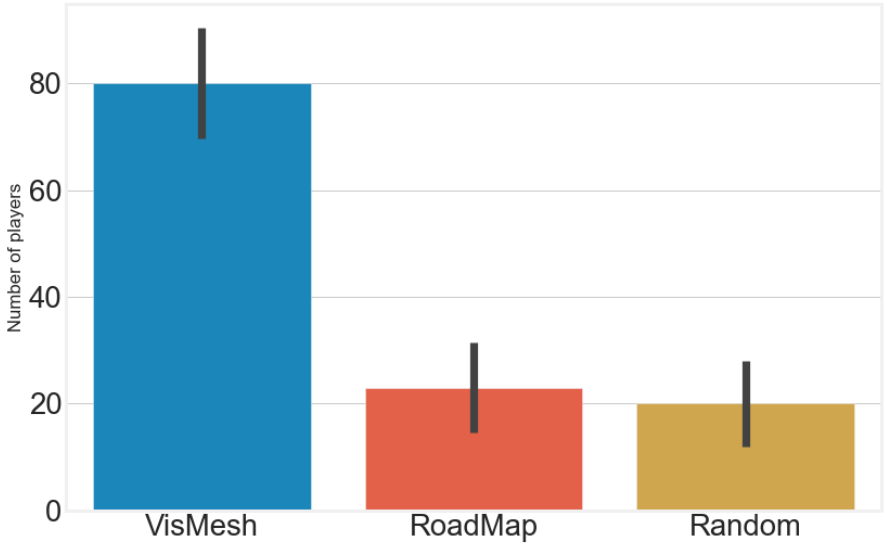


Figure 4.13: The number of players rating patrol behaviors as most challenging. The error bars represent a 95% confidence interval. [6]

Figure 4.14 presents a bar chart depicting participant selections of the most challenging team, organized by the sequence in which the teams were encountered during the study. Our analysis did not reveal any substantial influence of the order on their choice. However, an incremental pattern seemed to emerge with the VisMesh team, indicating that participants were more inclined to consider it the most difficult when they encountered it later in the study. It is important to note that these observations fall within the 95% confidence interval, rendering this result statistically insignificant, and further testing would be necessary to confirm its validity.

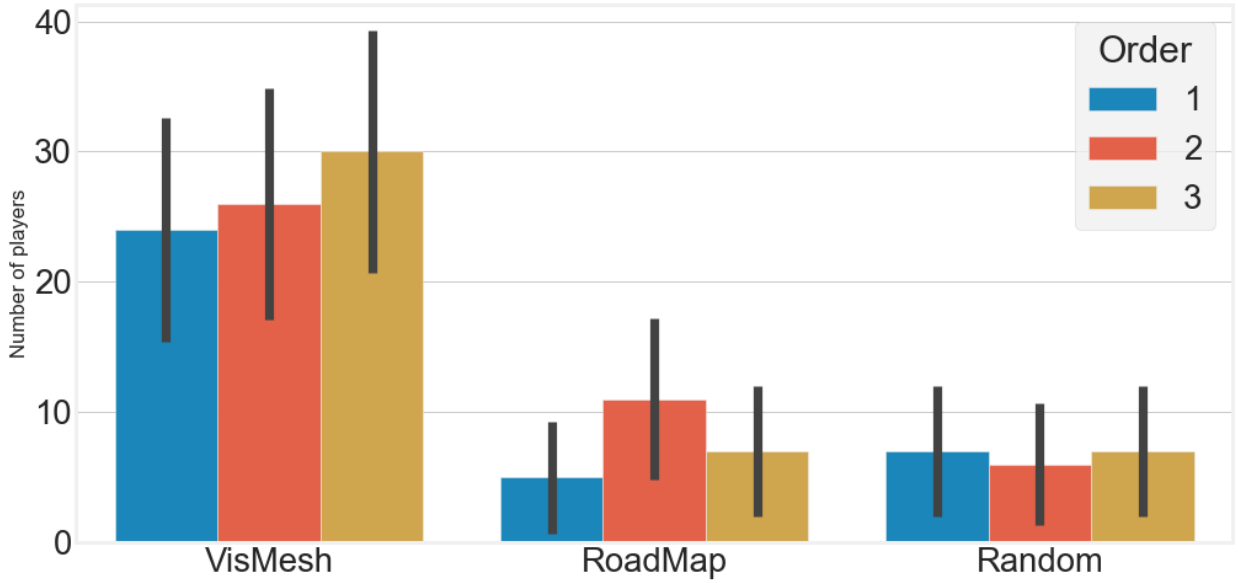


Figure 4.14: The number of players rating patrol behaviors as most challenging is grouped by the order in which the behavior appeared. The error bars represent a 95% confidence interval. [6]

Regarding the rationale behind their selection of the most challenging team, we identified the primary characteristics that participants cited as:

Meticulousness

Half of the players indicated that they found a team challenging when the guards systematically checked rooms and meticulously examined the corners within these rooms, resembling a thorough search. This behavior created an impression of the guards being highly conscientious in their patrol duties.

Collaboration

Participants pointed out that the team appeared to be well-dispersed, resulting in less overlap and more comprehensive coverage of the game space. This characteristic, along with the previously mentioned one, can be indicative of effective patrol behavior, declaring a potential link between player perception of challenging and effective AI behavior in this study.

Upon comparing the correlation between players' scores against the teams they considered challenging and effective, we identified a robust linear relationship (Pearson correlation coefficient = 0.83, p-value < 0.05) between these attributes.

Unpredictability

Unpredictability emerged as a common attribute that players associated with a challenging AI. Intriguingly, many of the players who made this observation also selected the same team as the most enjoyable. This suggests that players indeed appreciate a certain degree of unpredictability in their gaming experience.

4.4.5 Effectiveness

We asked the players how they rated the effectiveness of how well the guard covered the level, and their choices aligned with their rating of the guards' difficulty, which made Vismesh the most effective among the three behaviors.

4.5 Threats to Validity

This study exhibits several constraints. Initially, the sample size was relatively modest, being restricted to university students. Although the findings displayed distinct patterns in player enjoyment, we emphasize the significance of a larger sample size.

Additionally, relying on self-reported measures for data collection might have introduced response bias and recall errors, potentially compromising the accuracy and reliability of the data. The brief duration of the study constrained the exploration of player adaptability to guard behaviors over time. Furthermore, the absence of a static patrol baseline behavior impeded our ability to assess the influence of these dynamic behaviors on replayability.

Finally, the investigation focused on a top-down game perspective, possibly neglecting the varied effects induced by different viewpoints. Despite these limitations, the study imparts

valuable insights into dynamic patrol behavior, motivating further research to address these drawbacks and provide deeper understanding of its influence on player enjoyment.

4.6 Summary

This chapter presented the results of our user study that aimed to assess the impact of various patrol behaviors on player experience in a stealth game. According to the survey responses, the participants showed varying levels of enjoyment when playing against different patrol behaviors, with RoadMap and VisMesh being significantly more favored than Random patrol. Additionally, players significantly found VisMesh to be the most effective and challenging patrol behavior, exhibiting search-like behavior.

Several trends emerged from our study. Notably, many players derived enjoyment from VisMesh despite their poor performance. Analyzing their feedback indicated that the most difficult behaviors were the ones they found most motivating for enhancing their gaming abilities. Furthermore, players noted that the game enjoyment was heightened by the unpredictability of these behaviors, with VisMesh standing out as the most unpredictable patrol behavior. The confirmation of these patterns' significance warrants further exploration through future testing.

At the start of the patrol scenario, guards possess no presumptions about the potential presence or location of intruders. Nevertheless, in numerous games, once they detect an intruder, they initiate an active pursuit. If the intruder manages to elude their line of sight, the guard then adopts a search behavior, considering the last known location of the intruder. In the following chapter, we investigate this scenario and develop dynamic search behavior that factors in the layout of the game level.

Chapter 5

Guard Search Behavior

In the previous chapter, we explored how player experience is affected by the dynamic patrol behaviors we introduced. In this chapter, we extend the work of dynamic behavior to include how guards search for an intruder after being spotted.

Game mechanics often incorporate variations on hide-and-seek or pursuit/evasion, particularly in action and stealth genres. Players may need to evade observation by enemy NPCs as part of a game objective or to avoid combat; the latter is common even outside of pure stealth contexts. However, enemy NPCs subsequent pursuit and search behavior is often based on simplistic strategies: enemies may possess complete information about the player's position despite occlusion, which is perceived as unfair, or they may exhibit unrealistic search behavior, such as moving or looking randomly or only in pre-determined locations, which could potentially undermine the challenge.

This chapter proposes new strategies for enemy agent searching behavior. Our approach aims to develop a method that gives enemy agents an appearance of searching based on prior observation and spatial knowledge while ensuring a practical implementation efficient in real-time game context. Our designs are based on a geometric decomposition approach that takes advantage of the game level layout by dividing it into a grid or using reachability properties of a structured straight skeleton to facilitate efficient propagation of probable player locations. Each approach shows advantages and disadvantages concerning the search efficiency

in finding an adversary, the computational efficiency, and the implementation complexity. The grid-based approach is inspired by the commercial game "Third Eye Crime" [63]. That game modeled search behavior for NPCs using an *occupancy maps* structure. Occupancy maps are graphical representations used to model the occupancy or presence of objects or obstacles within a given environment, often used in fields such as robotics and computer vision [41]. Our roadmap approach is intended to overcome the limitations of grid-based methods, such as discretization artifacts and scaling issues. It ensures that the search behavior follows the overall shape of the space, resulting in better-guided location probabilities compared to random sampling approaches. Moreover, this method allows for effective separation heuristics, enabling multiple enemy agents to independently search while avoiding overlap, thus covering a space more efficiently. Another advantage of this method is that it can be applied to robotic contexts with tracking systems, as demonstrated in prior work [49].

In our study, we conducted experiments to evaluate the effectiveness of our proposed methods. We compared various parameter settings and our approach to an existing probability-based search model in multiple game levels created using Unity3D. Our findings indicate that the roadmap-based method had an acceptable performance in most maps for multi-agent scenarios. Additionally, we demonstrated that our method can be executed in real-time and thus can be deployed in real-time systems.

This chapter's significant contributions include:

- We present grid-based, and roadmap-based approaches for tracking the whereabouts of an adversary, which has potential applications in commercial video games.
- We empirically evaluated the method's performance on several maps modeled from currently existing commercial games.

The organization of this chapter is as follows. Initially, in Section 5.1, we explain the scenario employed for testing our methods. Subsequently, Sections 5.2 and 5.3 describe the search behaviors, namely grid-based and roadmap-based. Following that, in Section 5.4, we

outline our experimental setup to evaluate these methods. Lastly, in Section 5.5, we present and analyze our findings.

5.1 Scenario

To assess the effectiveness of our techniques, we modified the prototype, to begin with the intruder positioned in front of one of the guards so its initial location is known to all guards. This is to remove the lead time and randomness of a guard finding the intruder for the first time. To focus on the search behavior, we gave the intruder a velocity that was 1.5 times greater than that of the guards, enabling the intruder to quickly move out of the guards' FOV. An illustration of the starting point of the scenario is presented in figure 5.1.

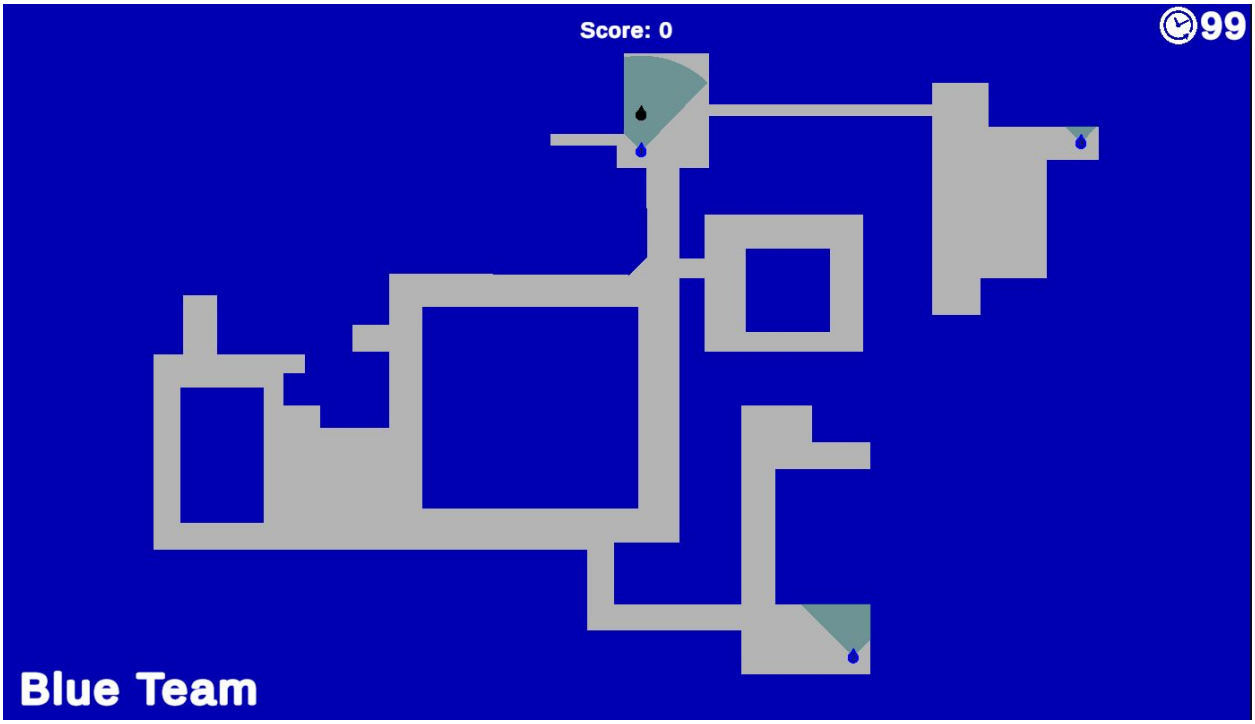


Figure 5.1: A screenshot of the beginning of the search behavior scenario. The map shown here is from Alien Isolation.

Once the guards lose sight of the intruder, they aim to locate the intruder again by searching the game level. Upon spotting the intruder again, all guards will become aware

of their location and take the shortest path to reach them in a simple chase behavior. The search behavior will restart if the intruder manages to evade the guards again, and so on.

There are multiple possible performance metrics for the search task: for example, the number of times the intruder is discovered after the first time. This provides the frequency of the intruder spotted by the guards; however, this is prone to inaccurately report search efficiency, such as in cases where the intruder goes around a corner while chased by a guard to be rediscovered again. To quantify the efficiency of the search behavior, we use the total time the guards are able to maintain visual contact with the intruder. Thus, if the guards successfully locate the intruder after it escapes their FOV, they will have more time to keep it in sight. This metric is referred to as the "alert time". This metric provides a continuous measure that overcomes the limitation of the discovery frequency.

The following sections will provide a detailed explanation of the methods we developed for search behavior. First, we will introduce a basic grid approach which is a variation of a technique used in a commercial game [63]. Then, we will describe how the roadmap structure can generate search behavior.

5.2 Grid

This method is inspired by occupancy maps, with the difference that instead of assigning to a cell a probability value representing the likelihood of the cell being occupied by an object or obstacle, it assigns the probability of an intruder being near that cell. Specifically, the game environment is divided into a fixed-size grid comprising cells categorized as walkable or non-walkable. Each cell is attributed a numerical value that denotes the probability of the intruder's presence in its proximity, referred to as the "likelihood". Guards can access this information and employ it to formulate their search strategy. The method encompasses two principal phases that alternate between each other: the representation-update phase, which updates the likelihood values of the grid to mirror the current state of the world, and the decision-making phase, where the guards plan their search actions to locate the intruder.

5.2.1 Representation-Update

While the intruder is in the FOV of at least one guard, all guards navigate to the intruder's position. However, when the intruder leaves the FOV of all guards, we assign a value of 1 to the cell closest to the intruder's last known position to indicate the likelihood of their presence in that particular location. The likelihood value is then spread to neighboring cells to account for the intruder's potential movement. We introduce two methods that determine the nature of likelihood spread.

Propagation

The suggested method propagates the likelihood to adjacent cells at a rate corresponding to the intruder's maximum velocity. This velocity-based adjustment guarantees that only the cells that are potentially accessible to the intruder are considered during the search, while those outside this range are excluded. Furthermore, the likelihood values of the remaining cells are gradually increased to reflect the time since they were last searched. Finally, to scale the likelihood values, we normalize them. The specific details of the propagation technique are described in algorithm 4, and figure 5.2 depicts an illustration of the likelihood propagation process.

This algorithm is executed per frame. It mainly spreads the likelihood omnidirectionally through the grid until it covers the whole grid. Then it slowly increments the likelihood value once the whole grid is propagated. First, it loops through each cell that has not been propagated and increments its neighbors' likelihood by a specific rate (line 8). Once its neighbors reach a likelihood of 1, that cell is considered to be propagated (line 15) and will be incremented by a much smaller rate instead (line 18). Finally, they are all normalized to restrict the likelihood values between 0 and 1 (line 24). This results in a domino-like effect of the likelihood of incrementing through the grid from the last position the intruder was last seen.

Algorithm 4 Grid propagation update

Require: $N_{walkable}$, the cells in the walkable area.
Require: I_S, the intruder's max speed.
Require: k, increment coefficient equals 0.001.

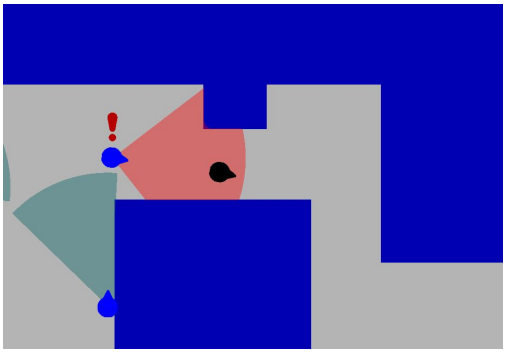
for each $n \in \mathcal{N}_{walkable}$ **do**
  if $\neg n.is\_propagated$ **then**
    $all\_neighbors\_expanded \leftarrow True$
    for each $neighbor \in n.neigbors$ **do**
      if $neighbor.is\_incremented$ **then**
        continue
      end if
      $neighbor.likelihood \leftarrow neighbor.likelihood + I_S * time\_lastframe$
      if $neighbor.likelihood < 1$ **then**
        $all\_neighbors\_expanded \leftarrow False$
      end if
      $neighbor.is\_incremented \leftarrow True$
    end for
    if $all\_neighbors\_expanded$ **then**
      $n.is\_propagated \leftarrow True$
    end if
  else
    $neighbor.likelihood \leftarrow neighbor.likelihood + I_S * k * time\_lastframe$
  end if
end for
$NormalizeLikelihoods(N_{walkable})$

Diffuse

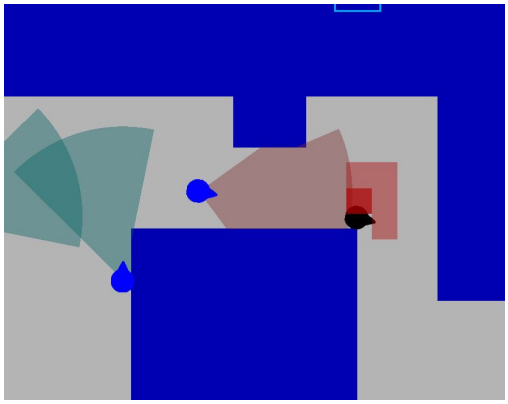
The second variation is inspired by Isla's implementation of occupancy maps in their commercial game "Third Eye Crime" [63]. At every time step, the likelihood is diffused to the neighboring cells using

$$P_{t+1}(n) = (1-\lambda)P_t(n) + \frac{\lambda}{|Adj(n)|}\sum_{n' \in Adj(n)} P_t(n') \tag{5.1}$$

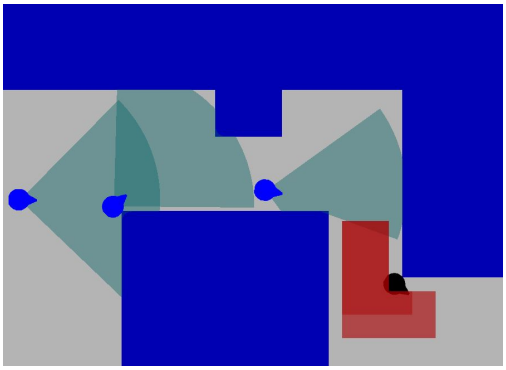
where $P_{t+1}(n)$ is the likehood of cell n after time step $t+1$, $\lambda \in [0,1]$ is the diffuse factor, *Adj(n)* are the neighboring cells to n. Figure 5.3 shows a scenario of how the likelihood is diffused. After diffusing the likelihoods, they are normalized over the grid.



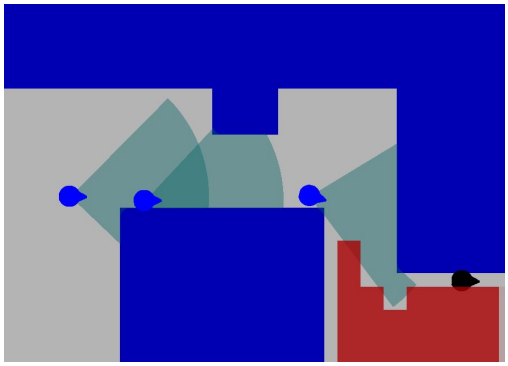
(a) The intruder, shown in black, starts in the center of one of the guard's FOV. The guards are the blue dots, and color of their FOV changes to red when they see the intruder.



(b) They move out of the guard's sight. The closest cell is set with a likelihood of 1, it start propagating to neighboring cells.



(c) Cells with non-zero likelihood are marked with red, and the higher the value the darker color.

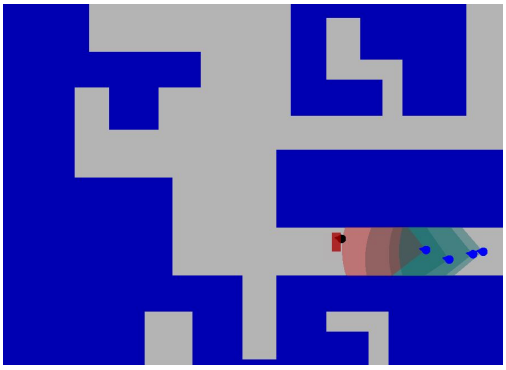


(d) Each guard starts searching by choosing a specific cell and find the shortest path to cover it with their FOV. By covering a cell, its likelihood is reset to zero. The cell selection depends on the decision-making settings of the algorithm.

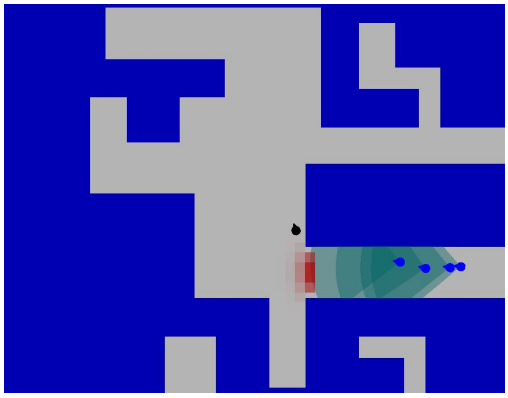
Figure 5.2: The steps of the grid propagation method. Once the intruder escapes the guards, the algorithm expands the likelihood omnidirectionally through the walkable space.

In a simplified explication, the primary difference between the two variations is how likelihood is moved. In the propagation variation, likelihood is spread across the grid in alignment with the speed of the intruder. This process approximates all possible positions

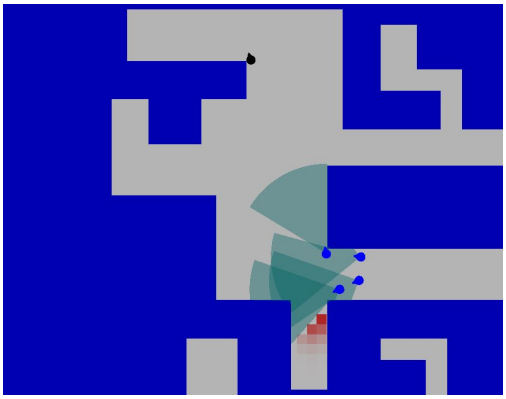
they could potentially inhabit. In contrast, the diffuse variation disperse the likelihood. This results in the immediate cell containing the highest likelihood, a value which subsequently diminishes as one moves towards cells more distant.



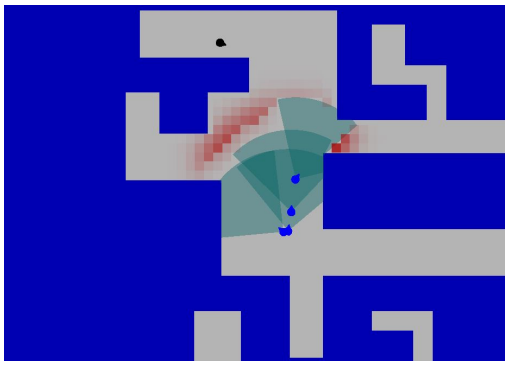
(a) The intruder (black dot) moves away from the guards (blue dots). Then the cell closest to the last known position is set with a likelihood of 1. Cells with a stronger tint of red have higher likelihood values.



(b) Being out of sight, the likelihood is diffused according to equation 5.1.



(c) This prioritizes the cells that are closer to the last place the intruder was seen. As depicted, the likelihood is highest in the dead-end where the intruder disappeared from the guards, as they had already covered the upper area where the intruder had moved.



(d) Once the guard covered the dead-end, the normalization rescale the likelihood other areas have higher likelihood and thus guards move toward it.

Figure 5.3: Illustration of the typical steps of the grid diffuse method, following Isla's approach.

5.2.2 Decision-Making

Similar to the decision-making process outlined in section 3.2.2, a cell to search is chosen for each unoccupied guard. Using the NavMesh, the guard determines the shortest path to the selected cell. After arriving at the target cell, the guard requests a new target. Typically, the guards aim to search the cell with the highest probability of being located near the intruder. Each guard chooses their next target cell based on its fitness value. The fitness value is a weighted average of the corresponding cell properties. Equation 5.2 shows how the fitness of a cell n is calculated.

$$\begin{aligned} f(n) = n.likelihood * w_{likelihood} + (1 - NormalizedDistance(n, g)) * w_{distance} \\ + DistanceClosestGuard(n, g) * w_{separation} \end{aligned} \tag{5.2}$$

The function $NormalizedDistance(n, g)$ computes the shortest path distance between a guard g and cell n. Additionally, the distance is normalized by dividing it by the game level diameter, which represents the shortest path distance between the two furthest points in the game level. The function $DistanceClosestGuard(n, g)$ returns the shortest normalized distance of the closest guard to cell n, provided that the guard is not g. The path distances are computed using NavMesh.

Furthermore, each weight ($w_{likelihood}$, $w_{distance}$, and $w_{separation}$) has a value between 0 and 1, and each one determines the relative influence of a specific property on the overall fitness of a cell to a guard. These properties include:

- Cell likelihood: The higher the value, the more probable the intruder is near the cell or has passed through it. The corresponding weight is $w_{likelihood}$.
- Path distance ($NormalizedDistance(n, g)$): To decide if closer cells are more desirable or vice versa. The corresponding weight is $w_{distance}$.
- Closest other guard distance ($DistanceClosestGuard(n, g)$): Improving coverage efficiency requires guards to reduce the time gathering. Selecting cells farthest from other

guards may offer better separation and improved coverage. The corresponding weight is $w_{separation}$.

5.3 RoadMap

Using a grid-based representation for the game level provides extensive coverage for locating the intruder, but it can be computationally demanding, especially for larger game levels with numerous cells. To overcome this drawback, an alternative approach involves implementing a roadmap similar to the one detailed in section 3.3. However, unlike the method described in that section, where all line segments have a uniform value for guards to search, the roadmap implementation assigns likelihood values based on the intruder's last known position, which then propagates through the roadmap accordingly.

5.3.1 Representation-Update

After the intruder moves out of guards' FOV, the closest node on the straight skeleton graph to the intruder is selected, and its connected segments are assigned a likelihood value of 1. After that, this representation is updated during the search in two potential variations.

Propagation

In this variation, the likelihood spreads through the roadmap segments at a speed slightly faster than the intruder's maximum speed. This is necessary to account for the structure of the roadmap since the NPCs use the NavMesh for pathfinding, which finds more efficient paths. One way to visualize the propagation of the likelihood through the roadmap is to think of it as a fluid flowing through a series of interconnected pipes. The likelihood flows through the network to indicate the potential locations the intruder could occupy. The propagation method is detailed in algorithm 5.

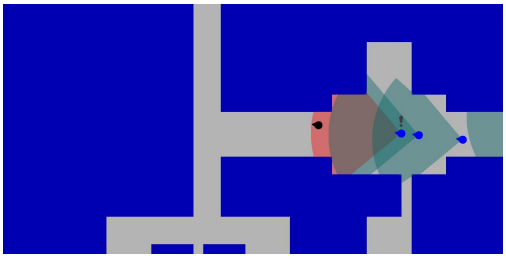
Each time the algorithm runs, it iterates through the segments; if the segment is expanded for its specified length on the roadmap, it propagates its likelihood value to the neighboring

Algorithm 5 RoadMap propagation update

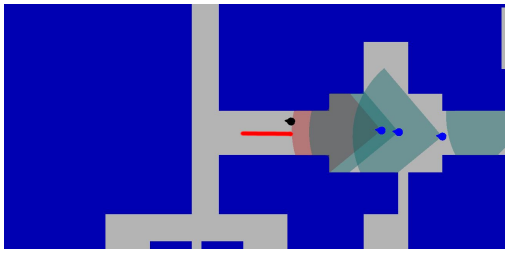
Require: R, the segments in the roadmap.
Require: $I_{MaxSpeed}$, the intruder's max speed.
Require: k, increment coefficient equals 0.001.

```
1: for each r ∈ R do
2:     if r.is_expanded then
3:         if ¬r.is_propagated_neighbors then
4:             for each n ∈ r.neighbors do
5:                 n.likelihood ← r.likelihood
6:             end for
7:             r.is_propagated_neighbors ← True
8:         end if
9:     else
10:        Expand(r, I_MaxSpeed)
11:    end if
12: end for
13: r.likelihood ← r.likelihood + time_since_lastframe * k
14: NormalizeLikelihoods(R)
```

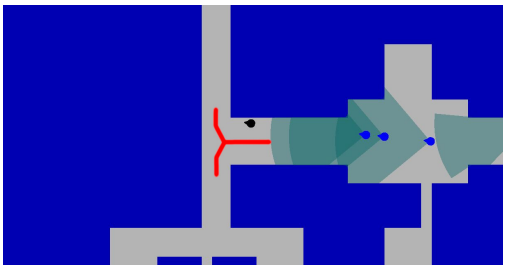
segments (lines 4-6). After that, if it is covered again by the guards, it will expand again till it reaches its maximum length. The $Expand(r, I_{MaxSpeed})$ function extends the segment to either match or exceed the intruder's maximum speed. Once a segment is fully expanded, if a guard sees part of it at any time, it will expand much slower. This is to make it easy for the guard to cover a whole segment, and the expansion speed matters little after the initial expansion. The likelihood will also increment slowly to keep track of the time a segment was covered (line 13). Lastly, the segment likelihoods are normalized. Figure 5.4 shows an example of the propagation variation.



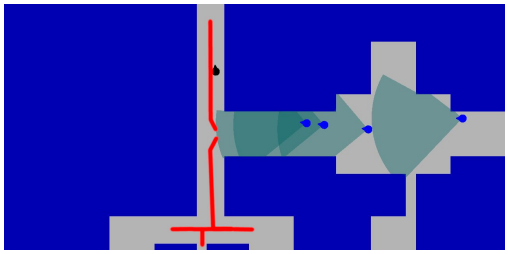
(a) The intruder starts in the center of one of the guard's FOV.



(b) Once the intruder is out of sight, the nearby segment is set with a likelihood of 1. The line segment with non-zero likelihood is shown in red.



(c) The likelihood flows in the roadmap, and as guards move, any part of the segment falls into their FOV is cut (to reflect that that part is seen) until completely covered, and their likelihood will be reset.



(d) The likelihood value is propagated through the roadmap, and so on.

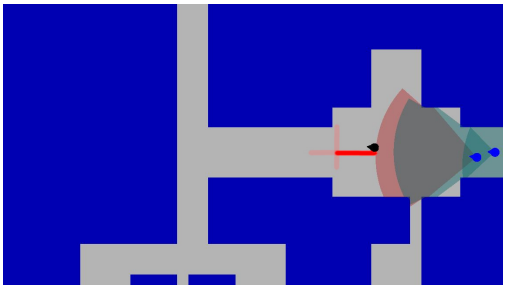
Figure 5.4: Illustration of the typical steps of the roadmap propagation method. As the intruder escapes, the likelihood flows through the roadmap matching their movement speed.

Diffuse

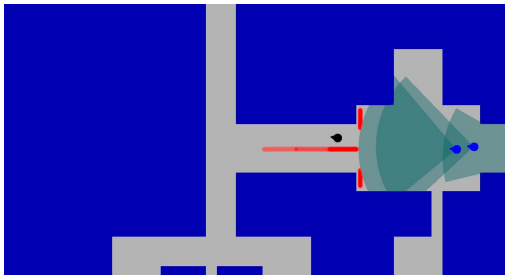
For diffusion, we took inspiration from Isla's implementation of occupancy maps in their commercial game "Third Eye Crime" [63]. In our implementation, instead of diffusing the likelihood in a grid, we accommodate the diffusion in the roadmap. So the likelihood will be diffused between the segments of the roadmap. At every time step, the likelihood is diffused to the neighboring segments using

$$P_{t+1}(r) = (1-\lambda)P_t(r) + \frac{\lambda}{|Connections(r)|} \sum_{r' \in Connections(r)} P_t(r') \tag{5.3}$$

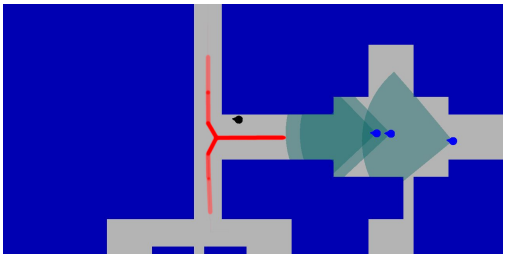
where $P_{t+1}(r)$ is the likehood of segment r after time step $t+1$, $\lambda \in [0,1]$ is the diffuse factor, *Connections(r)* are the directly connected segments to r. Figure 5.5 shows a scenario of how the likelihood is diffused. After diffusing the likelihoods, they are normalized over the roadmap segments.



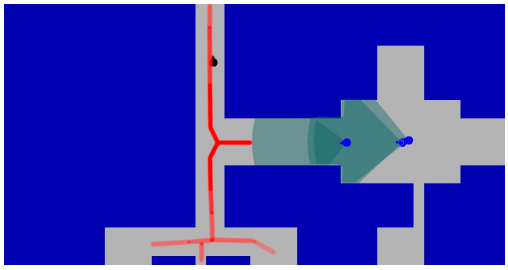
(a) The intruder moves away from the guards.



(b) Being out of sight, the likelihood is diffused according to equation 5.3.



(c) The segments will diffuse so the closest to the intruder's last location will have a higher likelihood.



(d) Once the guard cover the cells, the normalization rescale the likelihood so neighboring cells will have a higher value.

Figure 5.5: Illustration of the typical steps of the roadmap diffuse method. Unlike the propagation variation, the likelihood is diffused gradually through the roadmap.

5.3.2 Decision-Making

Every inactive guard will request a new action plan following the roadmap update. The representation permits two types of plans. The first plan entails selecting a segment and

determining its shortest path via the NavMesh as previously described. The second plan involves the guard creating a comprehensive route to cover a series of segments in sequence instead of only selecting a destination segment. The purpose of the latter plan is to consider the path taken by the guard rather than solely focusing on a single segment. Taking into account the entire path leverages the fact that the guard covers multiple segments along the way, thereby enhancing search performance by incorporating these segments into the planning process.

Choosing a Segment

Similar to the grid representation, we evaluate the fitness of a segment by computing the weighted average of its properties and assign the segment with the highest fitness to the next available idle guard. Equation 5.4 shows how the fitness is calculated.

$$\begin{aligned} f(s) = s.likelihood * w_{likelihood} + (1 - NormalizedDistance(s, g)) * w_{distance} \\ + DistanceClosestGuard(s, g) * w_{separation} \end{aligned} \tag{5.4}$$

These properties are similar to those described in section 3.3.2.

Building a Path

Following a similar approach for patrolling in roadmaps, which we described in section 3.3.2, the guards plan a complete path instead of just a destination segment. Upon a guard determining a path, the relevant line segments are updated such that when other guards plan their path, they select line segments with fewer or no guards traversing through them. This could promote the separation between guards at the destination and the whole path. Algorithm 6 details how we achieve path construction for a guard.

In summary, $GetClosestSegmentOnRoadMap(g, S)$ returns the closest segment in the roadmap to guard g (line 9); It represents the start segment of the potential path to be built. After that, similarly to the Dijkstra algorithm, we build a path by exploring the

Algorithm 6 Roadmap decision making for a guard

```
1: function GetPath(S,g) where S is the group of segments in the roadmap, g is the guard requesting a path.
2:     open_list ← {}
3:     best_segment ← null
4:     for Every segment s in S do
5:         s.path_utility ← 0
6:         s.path_parent ← null
7:         s.total_distance ← 0
8:     end for
9:     n ← GetClosestSegmentOnRoadMap(g, S)
10:    open_list.enqueue(n)
11:    while open_list is not empty do
12:        c_s ← open_list.pop()
13:        if c_s.total_distance >= MAX_DISTANCE then
14:            break
15:        end if
16:        for Every non-visited segment s of c_s do
17:            utility ← GetUtility(s)
18:            total_utility ← utility + c_s.path_utility
19:            if total_utility > s.path_utility then
20:                s.path_utility ← total_utility
21:                s.path_parent ← c_s
22:                s.total_distance ← c_s.total_distance + s.length
23:            end if
24:            open_list.enqueue(s)
25:        end for
26:        if best_segment.path_utility < c_s.path_utility then
27:            best_segment ← c_s
28:        end if
29:    end while
30:    if best_segment is not null then
31:        return GetSimplifiedPath(best_segment)
32:    end if
33: end function
```

path with the highest total utility. However, we stopped expanding the search if the total distance reached the defined limit (lines 13-15). We expand the search by iterating through the connected segments to the current segment and update the possible highest total utility it can reach along with the total distance to reach it (lines 16-23). After the search, we backtrack the path from the segment with the highest utility to the start (lines 30-32). The acquired path follows the roadmap, and we refine it utilizing the NavMesh by simplifying the path and navigating around corners while keeping its overall abstract trajectory.

Just like the corresponding method in section 3.3, a guard will select a path for their next plan based on its overall utility, which is the sum of individual utilities of the segments along that path. The utility of a segment "$GetUtility(s)$" is the weighted average of its properties. These properties are:

- MAX_DISTANCE: This is the max length of the constructed path. This is usually set as a percentage of the game level diameter.
- Segment likelihood: The likelihood of a line segment represents how likely the intruder is to be near it. A value of 1 indicates a high likelihood, while a value of 0 means the opposite.
- Portion of guards planning to pass through this segment: Each segment will store the number of guards planning to pass through it. We normalize the number by dividing it by the count of guards in the game level.
- Connectivity: This represents the count of segments linked to a specific segment. This attribute may prioritize paths that traverse highly connected segments. To normalize this value, we divide the number of connected segments by 10, which is determined as the maximum count of connections that a segment can have based on our preliminary testing of the considered game levels. Nevertheless, it is possible to compute this value at the beginning of the scenario for each game level.

- Passing guards threshold: This binary attribute impacts the value of $s_{guardPortion}$. In the first option, "ACTUAL", $s_{guardPortion}$ retains its originally defined value. In the second, "MAX", $s_{guardPortion}$ is set to 1 when one or more guards pass through the segment and 0 when no guards cross it.

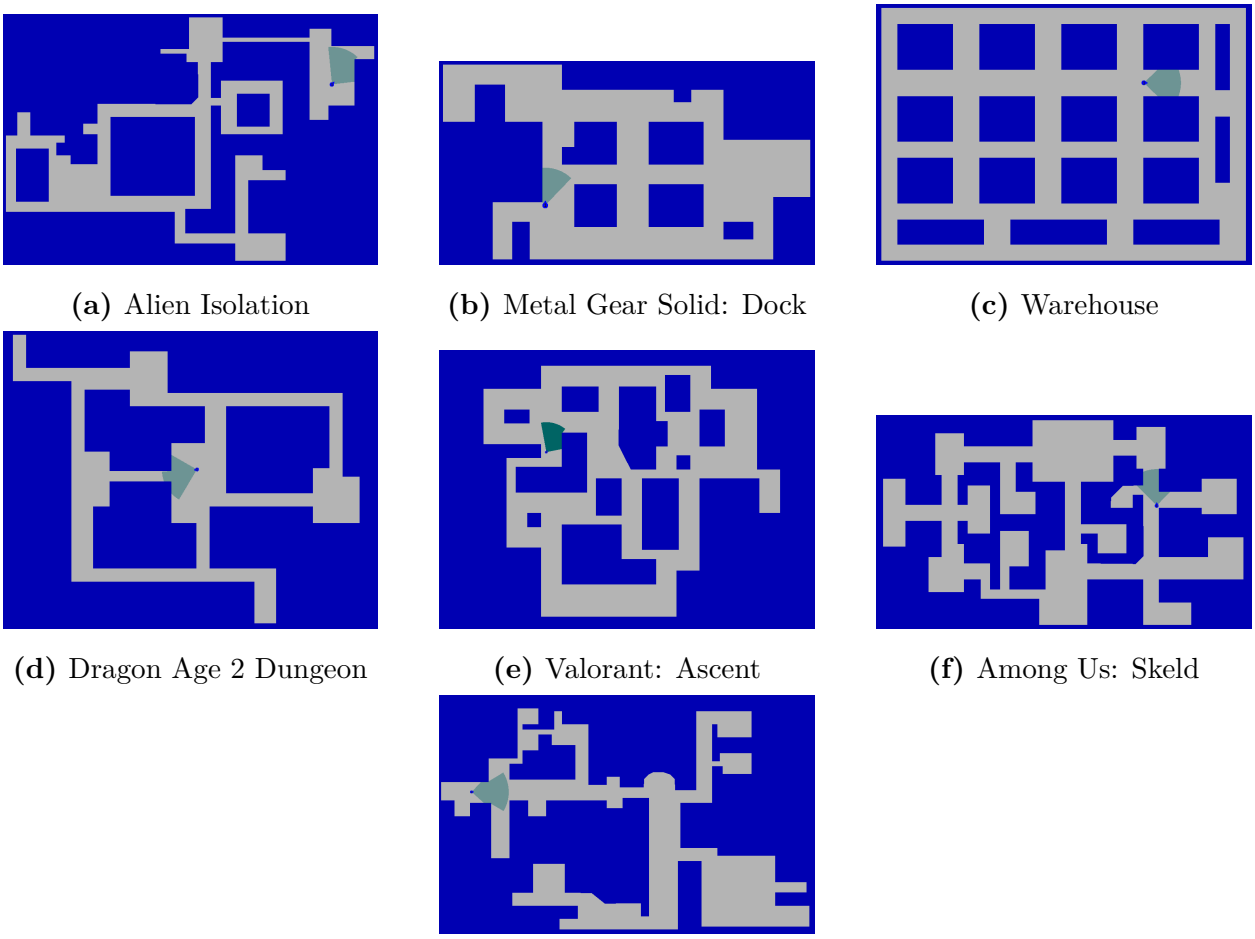
Equation 5.5 shows how the utility of a segment is calculated. The weights "w" for the corresponding properties are values set between 0 and 1.

$$GetUtility(s) = s_{staleness} * w_{staleness} + s_{guardPortion} * w_{guardPortion} + s_{connectivity} * w_{connectivity} \tag{5.5}$$

5.4 Experiment

This section outlines our method for evaluating and comparing search techniques based on their ability to track intruders and computational requirements. To achieve this, we conducted several rounds of the scenario described in section 5.1, varying the settings by combining the following elements:

- Game level maps: We incorporated the same maps used in section 3.6, and we show them in figure 5.6, to assess the impact of game levels on search performance. Appendix A provides additional information on their layout.



(a) Alien Isolation **(b)** Metal Gear Solid: Dock **(c)** Warehouse

(d) Dragon Age 2 Dungeon **(e)** Valorant: Ascent **(f)** Among Us: Skeld

(g) Arkham Asylum

Figure 5.6: The game level layouts included in this experiment.

- Number of guards: To examine the effect of the number of guards on search behavior, we considered each formation of guard teams consisting of three to six guards.
- Search method: We specify the included search methods and their settings in section 5.4.1.
- Intruder methods: The effectiveness of the search methods is influenced by the intruder's ability to avoid being detected. To explore this, we examine several intruder behaviors in section 5.4.2.

5.4.1 Search Methods

This section outlines the search methods and the parameter values assigned to them. We have utilized the parameters identified in section 3.5 as we believe that successful searching depends on effective area coverage. The search methods employed in our experiments are:

- Grid-based: This is method we defined in Section 5.2 with both of its variations of likelihood spread; the propagation and diffuse. We defined its parameters in table 5.1.

Table 5.1: The values chosen for the grid method.

Weight	**Value**
Cell dimensions	$0.5m \times 0.5m$
$w_{likelihood}$	1
$w_{distance}$	0.5
$w_{separation}$	0.5

- Roadmap: This is method we defined in Section 5.3 with both of its variations of likelihood spread; the propagation and diffuse. In addition, we include two variations of decision-making; segment-choosing and path-building. We set the parameters for the segment choice as in table 5.2. As for the path-building variation, the parameters are set according to table 5.3.

Table 5.2: The values chosen for the roadmap segment choice method.

Weight	**Value**
$w_{likelihood}$	1
$w_{distance}$	0.5
$w_{separation}$	0.5

- Random: For this experiment, we include two baseline methods representing two spectrum ends. The first is where the guards have no heuristic of the likely intruder locations. So each guard samples a random position on the road map and finds the shortest path toward it. Once the guard reaches it, it samples a new location, and so on.

Table 5.3: The values chosen for the roadmap path building method.

Weight	Value
Max Path Length	1
$w_{likelihood}$	1
$w_{guardPortion}$	0.5
$w_{connectivity}$	0.5
Passing guards threshold	Max

- Cheating: The second baseline method involves guards always being aware of the intruder's location. In response, guards take the shortest path toward the intruder, even if not within sight.

5.4.2 Intruder Methods

To evaluate our method, we need to test it against various intruder behaviors. For this, we simplified the problem by having the intruder choose a hiding spot from a predetermined set of positions on the map. We generated the set of hiding spots by assigning one spot on the bisector of a convex interior angle with a fixed distance angle and two hiding spots on the sides of an interior reflex angle with a small, fixed offset. Figure 5.7 displays the two scenarios.

(a) Reflex Angle **(b)** Convex Angle

Figure 5.7: Possible hiding spots based on the angle of the corners of obstacles.

Establishing effective hiding heuristics remains a research challenge in its own right [25, 101]. As such, we established three basic behavior patterns as a baseline for the intruder to choose their hiding spot:

- **Heuristic**: The intruder is conscious of the guards' locations and selects a hiding spot that has the highest path distance from the other guards. The intruder remains in that position until they are detected. Once spotted, they relocate and choose a new spot to hide.
- **Heuristic with Movement**: Rather than selecting one hiding spot and remaining there until discovered, this method enables the intruder to exhibit more complex behavior. Once reaching a hiding spot, the intruder waits for a random interval before relocating to another position, using the same heuristic regardless of detection status.
- **Random with Movement**: This behavior continually moves between hiding spots, waiting for a random time between moves. Rather than selecting the next hiding spot using the heuristic, the intruder randomly chooses the next location.

5.5 Methods Performance

To evaluate the search performance, we ran 20 episodes, each 99 seconds long, for different guard setups chasing an intruder for each combination of the search behavior, intruder behavior, and maps. We measure performance in relative "alert time", the proportion of time the intruder is observed versus staying hidden successfully. A lower alert time indicates an undesirable search performance.

This section aims to determine the optimal intruder behavior against all search methods, identify the most effective variations of the Grid and Roadmap search methods, analyze the performance of these methods in comparison to the baselines, and evaluate their computational efficiency.

5.5.1 Intruder Behavior

The effectiveness of search operations is considerably influenced by the intruder's ability to conceal themselves. A bar chart presented in figure 5.8 compares various intruder behaviors against all search behaviors. The chart suggests that intruders in motion were the easiest to find among all search behaviors, as they likely crossed paths with one of the guards. Conversely, intruders who remained stationary were more challenging to locate, as they had more opportunities to remain concealed. Consequently, our subsequent analysis primarily concentrates on static intruder behavior, given its substantial impact on search behaviors. In the next section, we explore the performance of the Grid method and its variations.

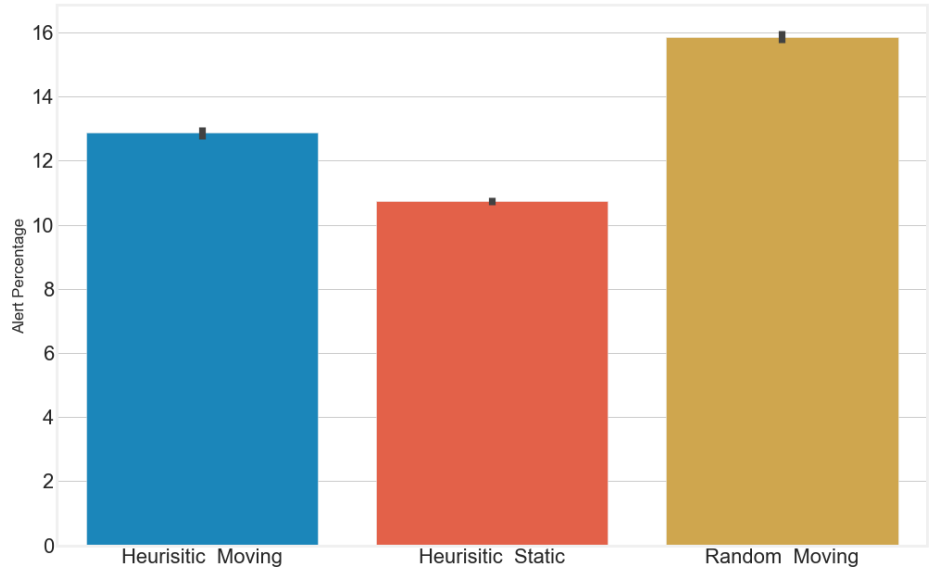


Figure 5.8: The overall percentage of time all search behaviors had the intruder under detection for the different intruder behaviors. The error bars represent a 95% confidence interval.

5.5.2 Grid

The bar charts depicted in Figure 5.9 illustrate the performance of the Grid method variations for all game level layouts. Our findings indicate that the "Propagation" variation

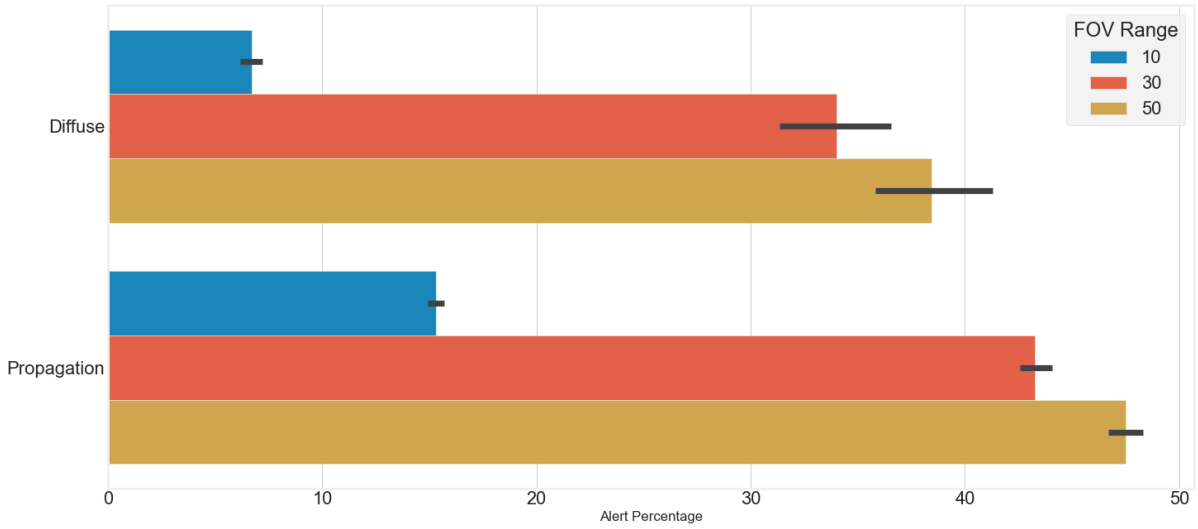
outperformed the "Diffuse" variation in all settings. This is primarily because the propagation method ensured a more uniform likelihood spread across the game level. In contrast, the diffuse method concentrated the likelihood in specific locations with a lower spread rate. Guards covered the search grounds in a breadth-first search by slowly extending the search radius, which resulted in guards grouping instead of spreading their search, negatively affecting overall performance.

By increasing the range of the FOV, the alert time significantly improved, making it more challenging for the intruder to hide. However, when we compared the impact of increasing the number of guards, we observed a slight but regular increment in the alert time for each added guard. Thus, while both can be used to adjust difficulty, regulating the number of guards seems to be a more precise way to adjust the difficulty level of the intruder's task rather than extending the guards' FOV.

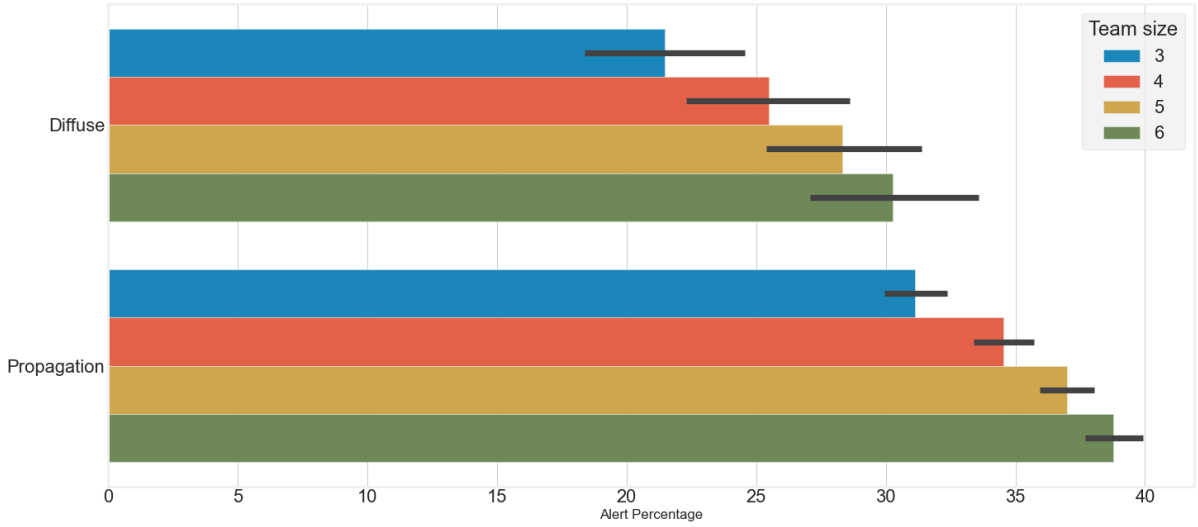
5.5.3 RoadMap

As for the RoadMap search method variations, the diffuse and propagation results are consistent with the Grid method. Figure 5.10 presents a bar chart to compare the RoadMap variations, it shows that the propagation variation outperformed the other variant. Additionally, we compared the form of the plan created on the search performance. Figure 5.11 shows a bar chart of the search performance for the two plan variations. In general, planning a complete path yielded better and more consistent performance. This is more noticeable as the size of the guard team increases and could be explained by the fact that the path-building plan considers the complete path rather than just the destination.

Building the complete path led to significantly more separation between the guards, which the results confirm in figure 5.12 that shows a bar chart of the total time guards spent within a distance of 1 meter of each other. In the next section, we compare all the search methods.



(a) FOV Range



(b) Guard Team size

Figure 5.9: The alert time percentage for the variations of the Grid search method. Each figure compares the effect of adjusting the guards' team size or the guards' FOV range. The error bars represent a 95% confidence interval.

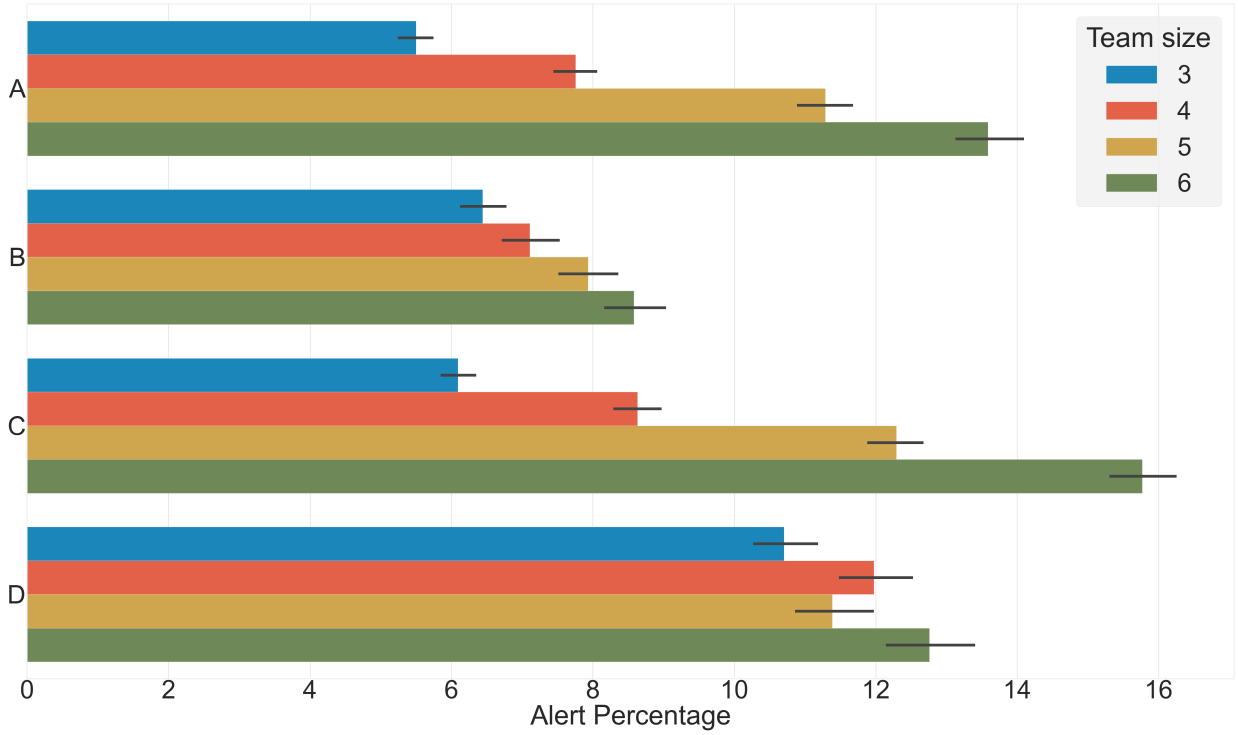
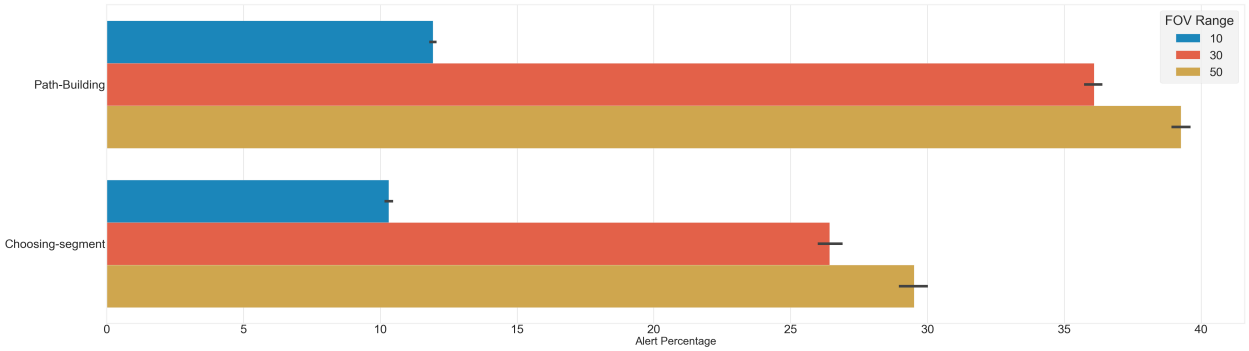


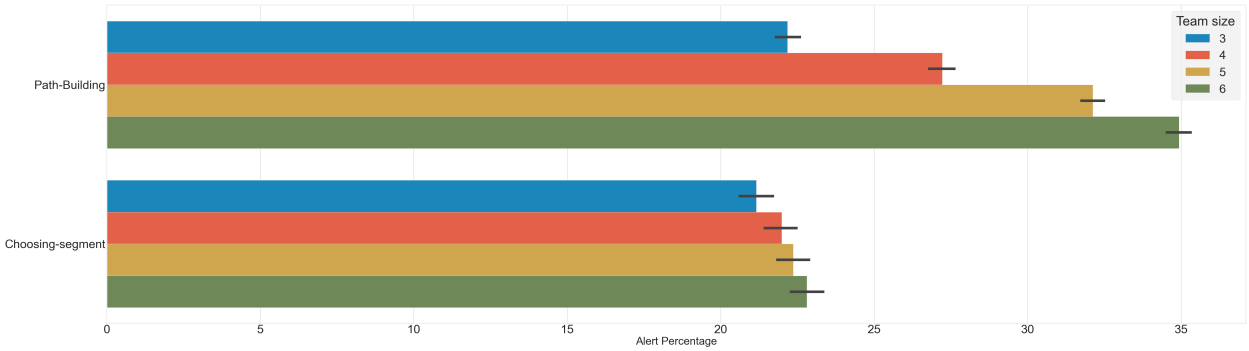
Figure 5.10: The alert time percentage for the variations of the RoadMap search method. The propagation variation outperformed the diffuse for all variations. The error bars represent a 95% confidence interval. A is the diffuse method using a path. B is the diffuse method choosing a segment. C, and D are for the propagation variation using a path, and choosing a segment.

5.5.4 Method Comparison

We compare the best-performing variations of the Grid and Roadmap methods with the two baseline search methods, Cheating and Random. We choose the propagation variation for the Grid method, and for the Roadmap, we choose the propagation and path-building variation. Figure 5.13 shows a bar chart of the alert percentage for the four methods over several team sizes with a FOV of 10%. We chose this FOV range because it highlighted the most difference in performance between the methods; as we extended the FOV, all methods converged to the same performance.



(a) FOV Range



(b) Guard team size

Figure 5.11: The alert time percentage for the variations of the RoadMap propagation search method. Each figure compares the effect of adjusting the guards' team size or the guards' FOV range, for both the path-building and segment-choosing variations. The error bars represent a 95% confidence interval.

The results demonstrate that the "Cheating" guards initially performed best. However, as the size of the guard team increased to 4-5 guards, the Grid method matched their performance. Furthermore, for six guards, the Roadmap method also achieved similar results to the cheating guards. In contrast, the two baseline methods did not exhibit a significant advantage when the team size was increased. Interestingly, the Roadmap method displayed the most significant improvement in performance with each additional guard. Compared to cheating, we believe that the primary reason for the efficient performance of the Grid and Roadmap methods is the effective dispersion of guards. Even if the guards took longer to search for the intruder, their dispersion allowed them to detect the intruder.

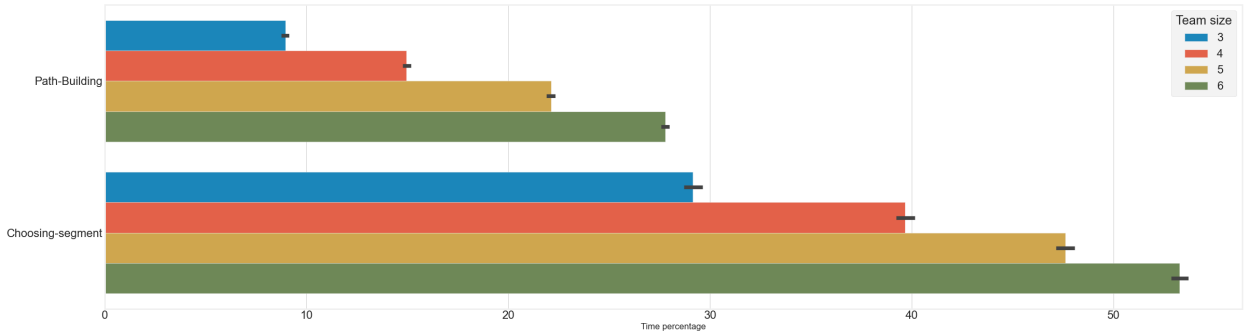


Figure 5.12: The overall percentage of time at least two guards spent close to each other within a distance of 1 meter during the scenario for the Roadmap propagation variations. The error bars represent a 95% confidence interval.

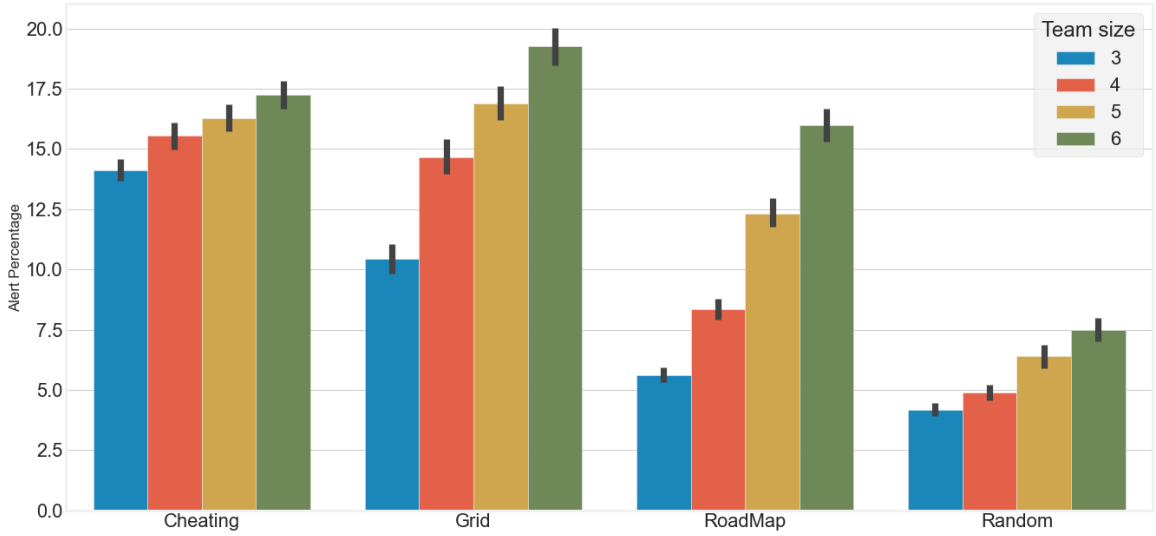


Figure 5.13: The alert percentage for all methods over different guard team sizes for all maps. All guards have a FOV of 10%. The error bars represent a 95% confidence interval.

To better understand how the map impacts performance, we examined the performance of guards with a 10% FOV on a team of five guards. We chose a team of this size as it fell within the middle range of our tests. Figure 5.14 presents a bar chart of the alert percentage for each method across different game level layouts. The results remained consistent across all game level layouts, with the Grid method performing almost as well as the Cheating guards, followed by the Roadmap method. However, the overall performance varied among

different game level layouts, possibly due to several factors, such as the game level's size, number of intersections, or map's complexity.

Additionally, the openness of the game levels could have impacted the search performance. As shown in Figure 5.14, the RoadMap exhibited lower performance in comparison to the Grid approach in "Arkham Asylum", "Metal Gear Solid", and "Valorant: Ascent". These specific game levels possess larger open spaces compared to the other levels. While multiple possible factors exist, our interpretation suggests that this contrast could be attributed to RoadMap's lack of adequate coverage within open areas. Consequently, guards were more likely to overlook the intruder when using the RoadMap, in contrast to the more inclusive coverage offered by the Grid approach. Addressing this issue can be a future research direction as it may necessitate the development of a more sophisticated roadmap designed to address these open areas.

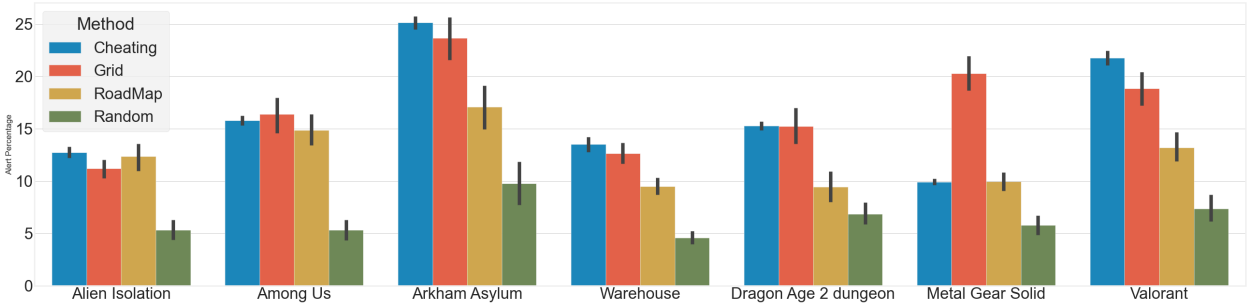
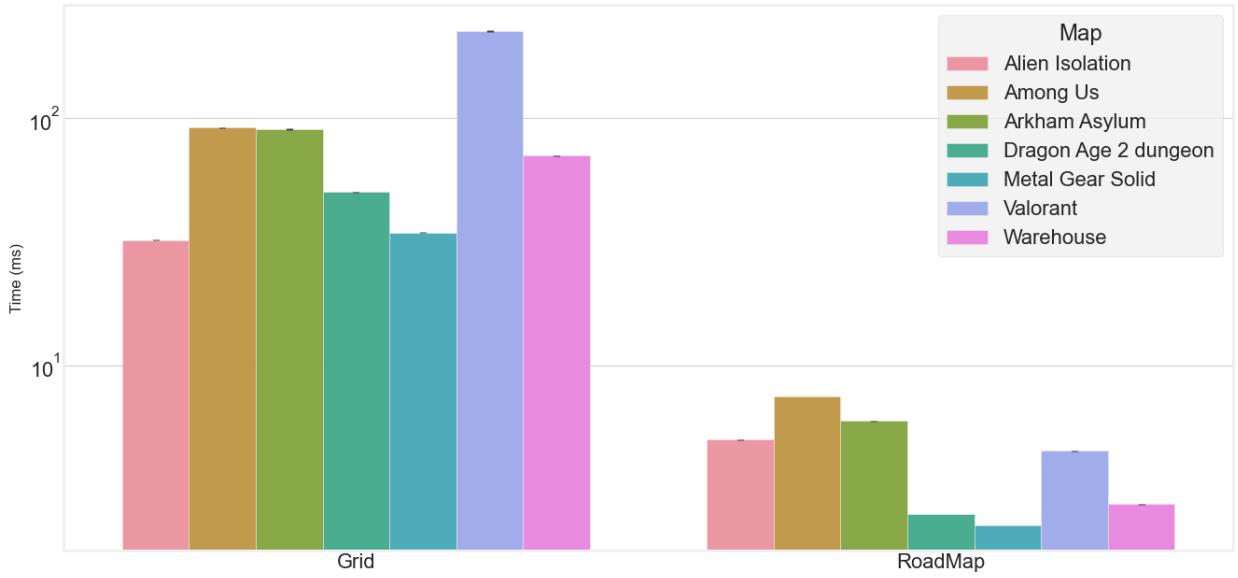


Figure 5.14: The alert percentage for all methods over different game level layouts. All guards have a FOV of 10%, and the team consists of five guards. The error bars represent a 95% confidence interval.

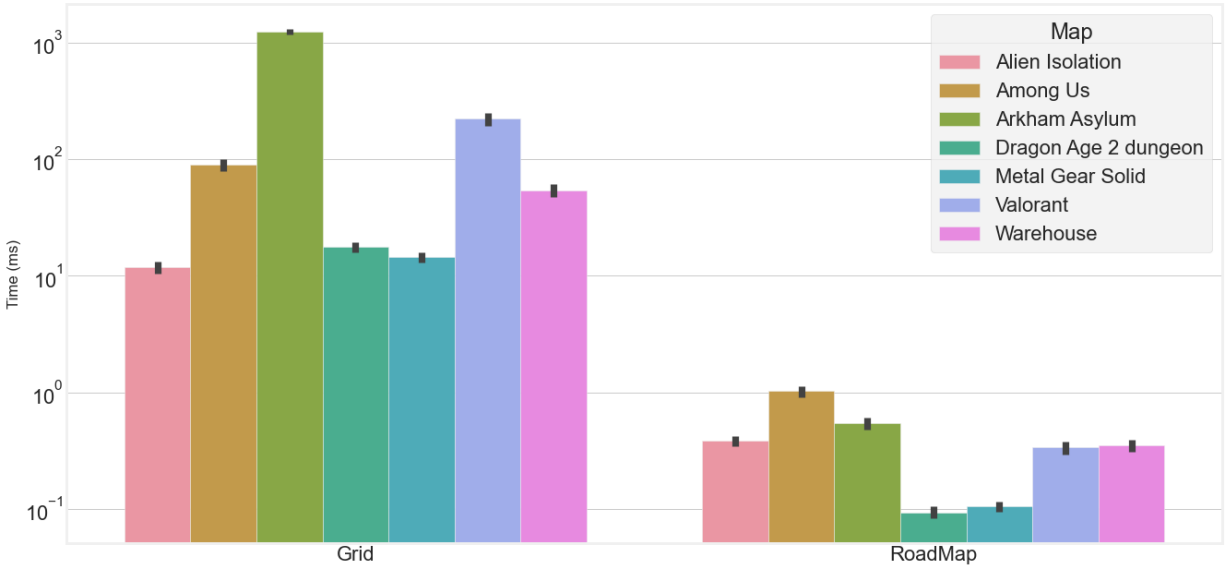
Results showed that the Grid propagation variation proved superior to the Roadmap performance. Still, for this method to be practically deployed in commercial video games, it must fit into the limited computational budget. In the next section, we compare the computational cost to compare the tradeoff between the Grid and Roadmap.

5.5.5 Computational Cost

It is crucial to evaluate the computational feasibility of these methods. As mentioned earlier, each method consists of two main steps: representation-update and decision-making. Figure 5.15 illustrates the time the methods take per iteration of these two steps for various maps. Our findings indicate that the RoadMap method is significantly less expensive than the Grid method, with an average update time of 2 milliseconds and a decision-making time of 0.3 milliseconds. This may be attributed to the sparse representation and straightforward decision-making technique employed by the RoadMap method. Conversely, we discovered that the Grid representation was the most demanding for all maps. The update step took an average of 70 milliseconds, and there was a much larger decision-making requirement with an average of 155 milliseconds. The high required time for both steps is likely due to the Grid method's maintenance of a larger number of cells in the Grid. One possible approach to reduce computation time is to utilize a coarser Grid. However, this may come at the expense of search performance, as we showed how a slightly coarser Grid affected the coverage performance of guards in section 3.5. Additionally, the cell size should be relatively close to the size of the NPC, which means larger game levels may still require more cells.



(a) Representation-update



(b) Decision-making

Figure 5.15: The computational performance for the Grid and Roadmap methods we defined for several maps. The y-axis is the time the method takes in milliseconds, and the range is set in the log scale. These results were gathered on an Intel® Core™ i5-7500 Processor CPU @ 3.40GHz, 32GB RAM, AMD Radeon R9 200 Series, Windows 10 machine.

5.6 Summary

In games, players tend to find NPCs that exhibit rational and intuitive behavior more interesting to play against. In this chapter, we presented a method that enables guards to demonstrate real-time search behavior for an intruder that more effectively utilizes awareness of the game level's geometry. We could propagate the probability of potential opponent locations using a skeletal graph representation. It demonstrated intriguing behavior for multi-guard searches for an intruder in several maps from existing commercial games. Our grid variation outperformed this method but matched it in relatively occluded contexts. The skeletal graph closely approximates the level shape and can be adjusted through various parameterizations, such as the extent of the guard's FOV. More importantly, it proved to be feasible to run in real-time scenarios making it more efficient in CPU time for larger maps.

In the next chapter, we will introduce a user study designed to assess player perceptions of dynamic search behaviors and explore how pre-defined verbal cues or "barks" announced by guards can influence these perceptions.

Chapter 6

Guard Search & Dialog User Study

In the preceding chapter, we presented a dynamic search behavior designed for guards to locate intruders once detected and managed to escape. In this chapter, we conduct a user study to assess player perception of the search behaviors and investigate the impact of employing various spoken dialogs by guards.

In the realm of gaming, it is well-established that designing an opponent AI that is exceptionally efficient or unbeatable does not always lead to enhanced player enjoyment [73, 124]. However, players do find pleasure in interacting with NPCs that exhibit intelligent or human-like behavior [103]. Therefore, it is desirable to incorporate elements of intelligent decision-making. Typically, game developers achieve this by either crafting intricate NPC behaviors that naturally indicate intelligence, while maintaining computational efficiency, or by artificially highlighting NPC decisions through visual or verbal cues, like having NPCs engage in dialogue that aligns with their perceptions or the game state [94]. This chapter dives into the relationship between these two approaches and their impact on the player experience. In essence, it seeks to determine whether it is more beneficial to invest in complex NPC behavior through algorithmic sophistication, which tends to require increased computation time, or if a simpler behavior can be augmented with dialogs to create the illusion of intelligence.

Our research is centered on a user study conducted within the context of a stealth-based video game scenario. In this scenario, players are tasked with exploring an environment while evading capture by opponent NPCs. The success of this endeavor relies on NPC intelligence, as they must effectively search the area and utilize their previous observations of the player's movements to apprehend them. This setup presents an engaging challenge for players, with the level of difficulty and the sense of fairness or realism hinging on how convincingly and naturally NPCs conduct their search.

One straightforward approach is to employ hard-coded search paths or "cheating" mechanisms, where NPCs possess knowledge of the player's exact location. While these methods are easy to implement, they can often come across as artificial. On the other hand, employing more intricate NPC AI that utilizes dynamic, localized information implies a greater investment in design and implementation effort, but this sophistication may not always be evident to the player. Both of these approaches can be enhanced by incorporating relevant NPC dialog that creates the illusion of intelligence.

We conducted a user study to evaluate the effect of NPC behaviors and dialog lines on player perception. The user study was based on playing the game we modified and survey data. We consider three forms of NPC behavior for searching for an intruder and two forms of dialog lines with different levels of contextual information. We divide the users of this study into two groups; each group compares a baseline search behavior against a heuristic and more effective behavior based on a relatively novel, geometry-aware path prediction from section 5.3.

The structure of this chapter is as follows. Section 6.1 provides a detailed account of the scenario in which players interacted with the search behaviors and dialogs. Subsequently, in Sections 6.2 and 6.3, we describe the search behaviors and dialogs themselves. Following that, Section 6.4 outlines the experimental setup employed to conduct the user study. Then, in Section 6.5, we present our results, analyze player responses and performance, and draw conclusions based on these findings. Finally, we describe the possible limitations of this study in Section 6.6.

6.1 Study Scenario

Each participant plays a four-round game, facing a unique team of guards distinguished by a specific color. We modeled each round similarly to our setup in section 5.1. Each guard team consists of four guards, each with a FOV that represents the scope of their visual senses. The round lasts for 99 seconds, during which the player controls a character to move around a game level. The guards' goal is to keep the player within their FOV, while the player's objective is to maintain their score by staying out of the guards' FOV. However, if at least one guard spots the player, all guards will chase them by taking the shortest path to their location. In addition, the player can increase their score by collecting randomly spawned coins located throughout the game level. Figure 6.1 shows a screenshot of what players see while playing the game.

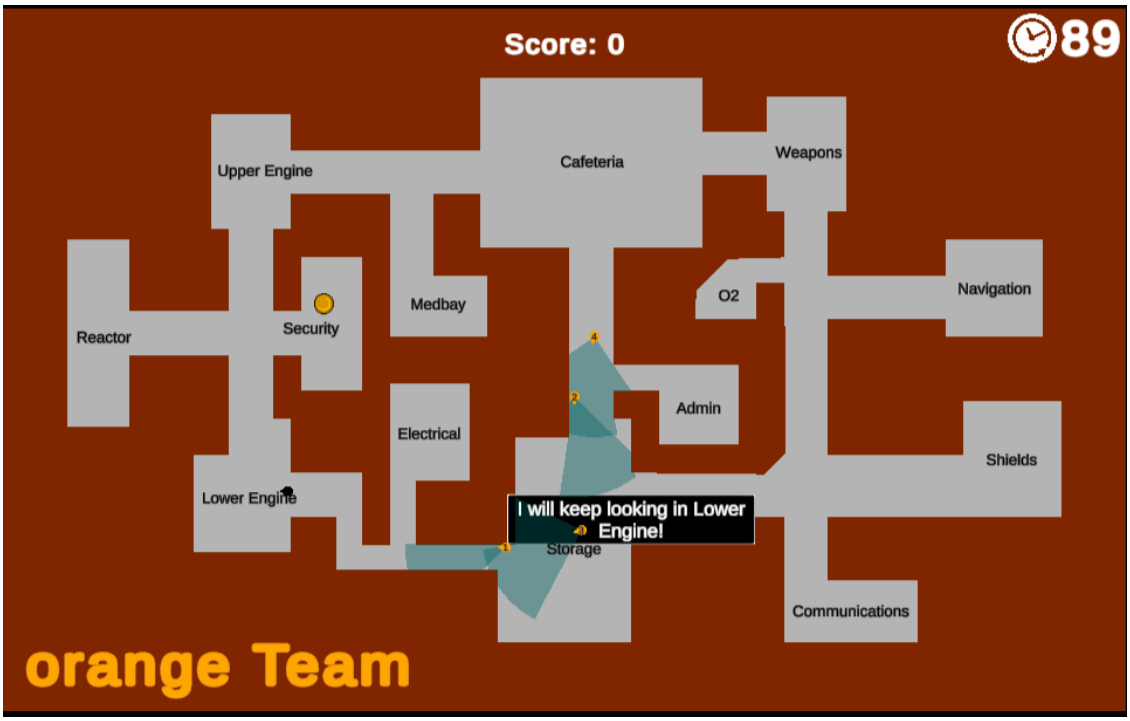


Figure 6.1: A screenshot of the game the participants played. The player (black dot) is in the "Lower Engine" room, while four guards are in the center area, each with a cone-shaped, translucent FOV. There is a coin in the "Security" room, which represents the player's goal. In addition, dialogue lines are announced visually and verbally to players using the Text-To-Speech function.

The scenario is similar to the one in chapter 5 with additional features and settings. The main elements are:

- Game level: The game level was a polygon with holes, where the walkable area was shaded in grey, and the unwalkable areas were colored with the same color assigned to the current team of guards. To ensure players can observe the guards' motion, we rescaled the level to allow the player to view the walkable area in a single frame, ensuring they had complete information on the game elements.

- Guards: Similar to our setups in the previous chapters, we chose a group of four guards with a 90° FOV and a limited range. These guards were randomly placed on the game level at the beginning of each round. Their primary objective was to pursue the intruder once they were detected in the FOV of any of the guards. The guards were all controlled by a central behavior manager. When the intruder is out of sight, the guards follow one of the search behaviors defined in section 6.2.

  Moreover, the guards were programmed to occasionally announce dialogue lines described in section 6.3. In a preliminary study, we found players to significantly enjoy playing against guards with simple barks rather than silent guards; hence in this study, we are interested in understanding how different dialog types would impact player experience. Each team of guards had a unique search behavior and a specific type of dialogue variation. To help players distinguish between the teams, we assigned a different color to each team. However, to eliminate any potential color biases, we randomized the color assignments for each session and the order of rounds.

- Intruder: This is the character the player controls, represented as a black circle. To give an advantage to players, we set the intruder's movement speed to be 1.5 times the speed of a guard. To reduce variation, the intruder is always randomly placed in front of a guard at the start of an round. To motivate players to move around the game level, we tasked the intruder with escaping and subsequently avoiding the guards while

also collecting coins spawned on the map. The intruder has a score, which increases when a coin is picked up, and gradually decreases when any of the guards spot them.

- Coins: After the player moves out of the guards' FOV, a coin is generated at the furthest corner from the player's current position. The player's score increases upon collecting the coin, and a new coin spawns in the same manner. If any of the guards spots the player, the spawned coin disappears and reappears only when the player moves out of sight. We defined this distance constraint to create a uniform challenge among the players and reduce randomness.

- Time: We set the duration of each round to 99 seconds, which we deemed sufficient for players to observe and form ideas of the guard behavior without exhausting their attention over multiple game-plays in our study. Additionally, we established a goal for players by allowing the game to finish once they attained a score of 100. Achieving this score remains challenging since players must collect at least five coins to reach it without being spotted.

In this study, we aim to evaluate the influence of two factors in guard behavior on player perception. The first factor is the search behavior that the guards employ to find an intruder, while the second factor is the level of information presented in the guard dialog.

6.2 Search Behavior

We have studied three search behaviors, each of which is adopted by a specific team of guards. These behaviors include a more advanced heuristic search approach and two simpler baseline behaviors.

6.2.1 Roadmap

This behavior represents a more advanced heuristic search method, which we explained in detail in 5.3. It uses the straight skeleton to project the possible generic paths the intruder

might take. Among the method's variations, we choose the propagation defined in 5.3.1 for the representation, and for the decision-making component, we choose individual segment search, which we described in 5.3.2. We chose this variation since it was the only one we developed at the time of this study.

6.2.2 Cheating

This is a baseline behavior where guards cheat by having the intruder's location at all times. All guards would take the shortest path to the intruder's current location. This method is considered cheating because knowing the player's location is not justified to players. This behavior is relatively common in commercial video games. For example, in the "Metal Gear Solid" series, guards will have information about the player's whereabouts shortly after being spotted, even after losing sight of the player.

6.2.3 Random

This represents the second baseline method, where guards follow a random movement pattern by uniformly sampling a random location to search. Each guard independently requests a random position within the walkable area and then takes the shortest path towards it using the VisMesh. This approach completely lacks knowledge of the player's whereabouts, making it distinctly different from cheating behavior.

6.3 Dialog

As a second factor of NPCs behavior, we aim to examine the impact of the level of information communicated by the guards through spoken dialogs that reveal their knowledge and intentions. In commercial games, NPCs often use spoken dialog lines for various purposes, such as drawing the player's attention to important events, adding a human-like element to characters, or enhancing storytelling. Our focus is on the dialog lines that NPCs use to announce their intentions to overtly reveal the decisions and reasoning made by the NPCs. In a

small preliminary study, we found that players significantly enjoyed when guards spoke with dialogs. Therefore, we believe that the mere existence of dialog gives players a stronger sense of NPC intelligence. Additionally, dialog lines can be particularly valuable in this case since the decision-making process of guards can be opaque to players. It simplifies this aspect, as players do not always have enough time to establish a cause for the guards' decisions.

During the chase or search for the player, guards in our game shout out dialog lines chosen from a set of lines appropriate for the current context. Our implementation is based on similar systems found in games such as "The Last of Us" and "Left4Dead" [48,96]. These lines are used to indicate when guards have spotted the player, lost sight of the player, or when they intend to search a particular area.

For our study, we defined two main variations differentiated by how much information they convey to players. Different levels of information suggest different guard reasoning and may thus affect how intelligent a guard appears, impacting a player's performance or enjoyment. The dialog variations are described below.

6.3.1 Abstract

Our abstract form is a moderately neutral form of dialog that conveys a guard's observations and plans without explicitly mentioning locations in the game level. Each spoken line has a set of preconditions that must be satisfied before announcing. These preconditions can be the guard's current state, the last timestamp the intruder's location was known, and the speaker's planned path distance. According to these preconditions, a dialog can be classified into these subcategories:

Spotting the Intruder

This is when a guard spots the intruder; based on how long they have been searching for the intruder, they would choose a line that reflects how long it has been. Table 6.1 shows several examples of these lines. To avoid repeating the same line in the same situation, we

Table 6.1: Abstract lines a guard can use on spotting an intruder.

Precondition time_since_intruder_seen (t)	**Lines**
$0 \leq t \leq 40$	**Over there!** **Here they are!** **Through here!** **Enemy sighted!** **Here!** **On me!** **I see them!**
$t > 40$ **seconds**	**I finally see them! Come here!** **They are still here! Everyone come here!**

included several lines so guards could rotate between them. Each time we search for a line, we search among the list in descending order according to the number of preconditions.

Announcing Intentions

Guards can announce their next move; however, for the abstract dialog type, lines are generic and do not reflect concrete plans. Table 6.2 shows examples of these lines.

Table 6.2: Abstract lines a guard can use to announce their intentions.

Preconditions **speaker_path_distance** (p) **search_elapsed_time** (t)	**Lines**
$p > 30\%$ **of the game level diameter** **and** $t < 2.5$ **seconds**	**I'll go around from the other end!** **I'll ambush them!** **I'll take the longer path**
$p < 20\%$ **of the game level diameter**	**I will search around!** **They must be nearby!** **I still need to check the nearby** **I need to check this corridor!** **I'll check this hall!**

Filler Lines

As guards search for the intruder, they announce lines that may add more human-like features. They do not announce clear intentions but observations or opinions that the player

might already know. Additionally, guards can reply to each other by replying with a line from a set of lines grouped as a reply for the announced line. Table 6.3 shows an example of a group of lines and the group of corresponding replies.

Table 6.3: An example of two sets of filler abstract lines. The first row is a line initiated by a guard, and the second is a set of replies a random guard can respond to.

Preconditions **search_elapsed_time** (t)	**Lines**
$t > 40$ **seconds**	**I still can't find them** **They're good!** **I'm tired of searching!** **It's like they vanished!**
Replies	**Yeah! but we have to find them!** **You can say that again!** **Yeah! We need to do better!** **I'm sure they're still here!**

6.3.2 Contextual

The second list of dialogs conveys more contextual information to players; the main difference is that the lines are associated with specific locations on the game level. In our study, rooms in the game level are labeled with names, like "Cafeteria", "Storage", etc. All rooms can be seen in the screenshot from the game in figure 6.1. This dialog group has the same subset of lines as the abstract dialog group. Table 6.4 shows a small set of contextual lines.

6.4 Experiment

To enroll individuals in this study, we established an online portal that offered a web-based version of our game. The majority of participants were recruited through a mass email sent to McGill University undergraduate and graduate students. We collected anonymous survey responses for participants' perceptions of the game along with their gameplay data.

The primary objective of this investigation is to assess whether participants can distinguish between a basic behavior and a more complex one, as well as identify factors that

Table 6.4: Examples of lines for the different subgroups of contextual dialogs.

Type of lines	Lines
Spotting the intruder	**They are in {intruder_last_seen_room}!** **On me in {intruder_last_seen_room}!** **Through {intruder_last_seen_room}!** **In {intruder_last_seen_room}!** **Come to {intruder_last_seen_room}!** **Alert in {intruder_last_seen_room}!**
Announcing intentions	**I need to clear out {speaker_goal_room}!** **They might be hiding in {speaker_goal_room}!** **I will keep looking in {speaker_goal_room}!** **I'll keep looking in {speaker_goal_room}!**
Filler lines	**I'm checking {speaker_goal_room}! You?** **{speaker_goal_room}! And you?** **In {speaker_goal_room}... you?**
Filler lines - Response	**Ok! I'm going to check {speaker_goal_room}!** **I'm going {speaker_goal_room}!** **Good! I'm going to {speaker_goal_room}!**

might impact their perception. To accomplish this, we assigned participants randomly to two separate groups. The first group compared guards utilizing the Random and RoadMap search behaviors, while the second group compared guards using the Cheating and RoadMap search behaviors.

At the beginning of each session, participants were given a choice to play a tutorial round featuring two guards utilizing Random search behavior. The game map was modeled after one in the Metal Gear Solid video game, as depicted in figure 6.2. To keep things simple, guards had no dialogue at this level. This tutorial round aimed to help participants become comfortable with the game's mechanics. Before beginning the actual study session, participants could replay the tutorial level as often as desired. However, our results showed that players rarely chose to replay the tutorial level.

To limit the study time and avoid overwhelming the players, we focus on making them compare a complex behavior and a simple behavior with the different dialog types. Players always compare a complex behavior and a simple one to see if they can recognize the more advanced behavior. Thus, after the tutorial round, each participant had to play four rounds.

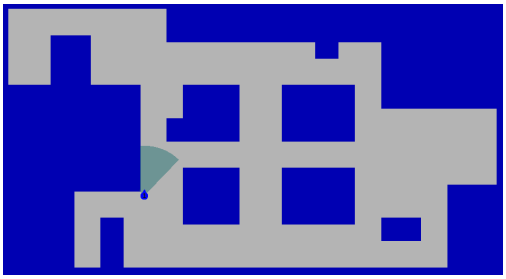


Figure 6.2: The Docks map from the commercial game Metal Gear Solid [68]

In each round, they played against a team of guards with a unique combination of one search behavior and one type of dialog group. Table 6.5 shows the format of each guard team of the four rounds. To get more representative results, the order of these rounds is randomized. We allocated a unique color to each guard team to aid participants in identifying them. These colors were randomly assigned to avoid any prejudice associated with colors or behaviors.

Table 6.5: The guard team assigned search behavior and dialog type. Based on the participant group, the basic search could be either the Random or the Cheating search behavior, and the Heuristic search is the RoadMap. The order for each participant is randomized.

Round
Basic search & Contextual dialog
Heuristic search & Contextual dialog
Basic search & Abstract dialog
Heuristic search & Abstract dialog

Regarding the game-level layout, it must present a certain level of difficulty to both participants and guards. A high degree of connectivity, with numerous pathways between different locations on the map, can make it easier for participants to elude the guards by maneuvering around barriers. Conversely, an excessive number of dead-ends can make it challenging for participants to escape. To address this, we opted to utilize the "skeld" map from the game Among Us. This map possesses several beneficial features, as it does not disproportionately favor one structure over another. It has a limited number of major cycles, allowing participants to access any location through multiple pathways. However, it is arranged as a series of rooms, most of which are dead-ends, providing additional difficulty

for participants. To improve the map's recognizability, we labeled specific locations with names to allow participants to associate them with particular regions. Additionally, to ensure participants could observe guard movements at all times, we fitted the entire map into a single frame, which is visible in figure 6.1.

To assess the participants' initial impressions, we posed the questions displayed in table 6.6 following each round. For each query, respondents could select a discrete answer from a predetermined list of options. We utilized a basic Likert scale consisting of three responses; the goal of using a small Likert scale is to simplify the rating task for participants. A screenshot of the question presentation can be viewed in figure 6.3. After the participants played all four rounds, we asked them to choose the team they enjoyed playing against the most and the team they found to be the most difficult.

Figure 6.3: A screenshot of an end-of-round survey question.

Table 6.6: The questions asked at the end of each round.

Question
How much did you enjoy playing against {team color}?
How hard was it to play against {team color}?
How natural the {team color} team's behavior was?

6.5 Results

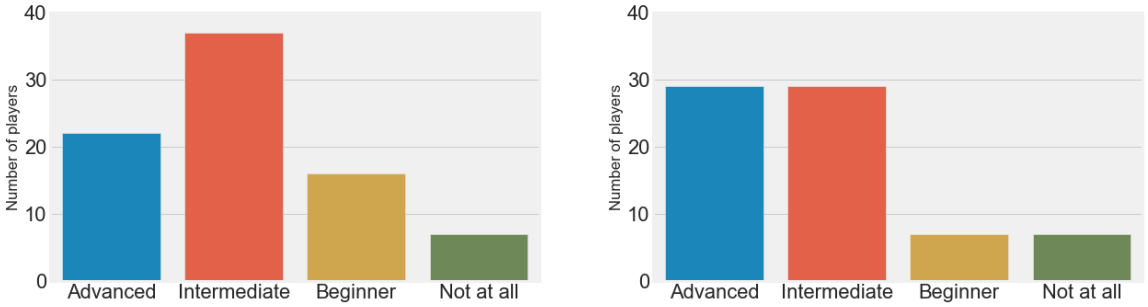
This section describes the study findings and discusses the impact of guard behavior and dialog type on the participants' experience. We organize this section by describing the participation statistics and the effect of experience level on the scores. We also investigate if participants performed better at later rounds to reduce the possibility of an evident learning curve. After that, we examine the impact of the guard search behavior and dialog type.

6.5.1 Participation

Our study involved 154 participants who completed all aspects of the study. Each participant engaged in a series of four rounds, each round featuring a unique combination of two dialogue variations and two distinct guard behaviors. To focus on the comparison between a complex and a simple behavior, one of the guard behaviors was set as the heuristic approach, while the second behavior was randomly selected from either cheating or random behaviors. This allocation effectively split the participants into two distinct groups. In the first group, consisting of 72 participants, individuals compared the heuristic guard behavior with the cheating guard behavior. The second group, comprised of 82 participants, focused on comparing the heuristic guard behavior with the random guard behavior. These participants contributed gameplay data and survey responses regarding each comparison.

As per our previous study in chapter 4, we requested participants to assess their familiarity with video games at the start of the study. The rating distribution of participants' self-perceived video game skills is presented in Figure 6.4. Notably, approximately 70–80% of the participants considered themselves to possess either advanced or intermediate proficiency in playing video games. This suggests that their findings may offer valuable insights into player perception.

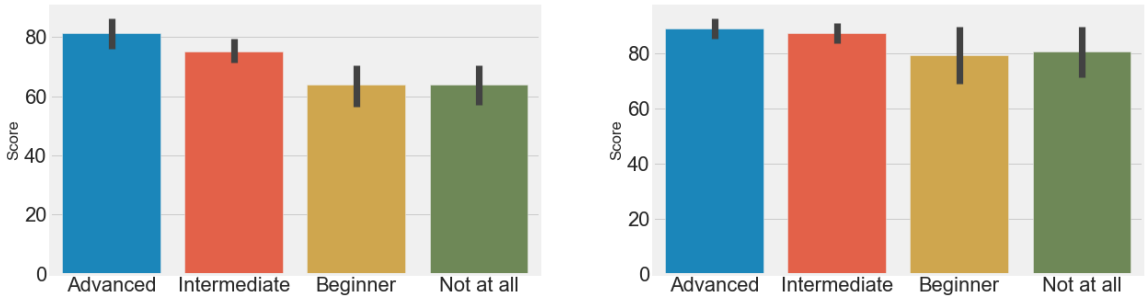
Despite the simple design of our game, we wondered if the participants' familiarity with video games could impact their performance, thus giving advanced players an edge over the other groups. Figure 6.5 shows no significant differences in the scores between the



(a) "Random" behavior group **(b)** "Cheating" behavior group

Figure 6.4: The distribution of how players rated their experience in video games for both experiments. The number of participants in the "Random" group, which consisted of the participants comparing the heuristic and random guard behavior, is 82, and the number for the other group is 72. [7]

different experience levels. Additionally, by comparing the scores over rounds in a preliminary result, we found that the scores remained somewhat consistent as participants progressed through the rounds. However, the low participant count can mainly explain the high variance observed in the lowest two experience levels (no experience and beginner). In general, we can deduce that the game had a low complexity making it possible for participants to learn the game mechanics in the early rounds.



(a) "Random" behavior group **(b)** "Cheating" behavior group

Figure 6.5: The scores participants achieved in the study. The participants sorted themselves into one of four experience levels in video games. The error bars represent 95% confidence intervals.

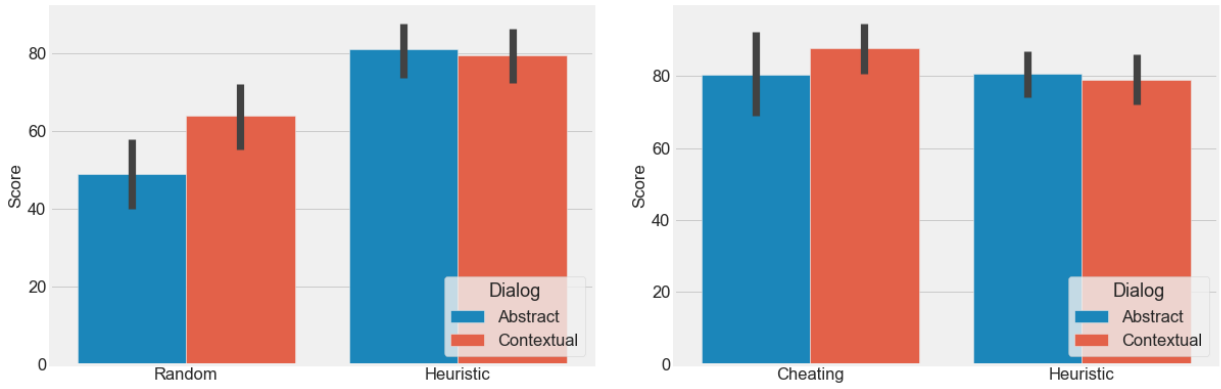
6.5.2 Performance

The way guards move is determined by the search behavior assigned to them. Participants could see how guards moved at all times of the game. Figure 6.6 shows that participants in the "Random" behavior group scored better when they played against the heuristic method. In contrast, the other group participants had similar scores between the "Cheating" and the heuristic behavior. Multiple causes can explain these results; even though randomly moving guards had no belief of the intruder's whereabouts, they were hard to predict and had more dispersion in the game level, while heuristic guards were relatively more predictable, which was easier to score against. As for the "Cheating" guards, even though the cheating guards had the intruder's location at all times, participants reported that they could use the cycles in the game level to their advantage by going around the guards in a circle while collecting the coins.

As for the dialog types, both types express guards' beliefs or intentions in the game; however, the main difference between the two types of dialog is the level of the conveyed contextual information related to places in the game level. To check if there was an impact on participant performance caused by the dialog types, we compared the scores they got in the rounds. Figure 6.6 showed no significant difference in score for both participant groups. However, contextual dialogs seemed to help participants score more than abstract in the "Random" group. This group competed against the more unpredictable guard behavior, so contextual dialog could have helped participants improve their performance against the guards by understanding the guards' plans.

6.5.3 End-of-round Ratings

As for the immediate responses the participants had after playing each round, we found them to enjoy the randomly behaving guards with abstract dialogs, but no significant differences in their opinions comparing the random and heuristic behavior in terms of difficulty and naturalness. On the other hand, participants had strong opinions regarding the natural-



(a) "Random" **(b)** "Cheating"

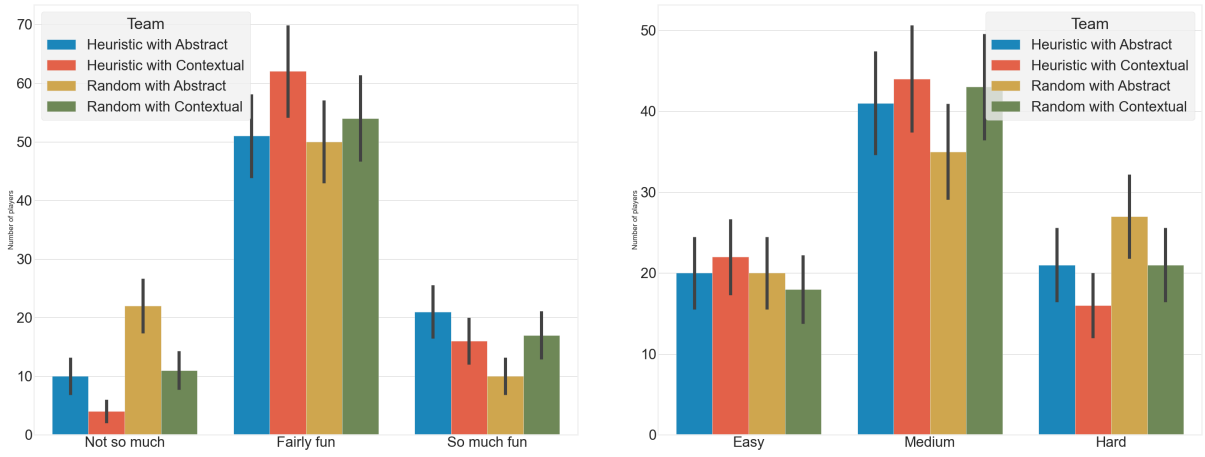
Figure 6.6: The scores participants achieved by playing the game for the two study groups. The error bars represent 95% confidence intervals. [7]

ness between cheating and heuristic behaviors. Figures 6.7 and 6.8 show the participants' responses after each round they played. They found the cheating to be unnatural, while heuristic to be very natural. This difference is much less evident when compared to random behavior. This shows that participants could understand the underlying behavior of cheating guards during the round; however, fewer participants seemed to dislike the random behavior over the heuristic in the other group.

In the "Random" group, participants enjoyed the contextual dialogs. While in the "Cheating" group, figure 6.8 shows that the contextual dialog interestingly gave participants a sense of difficulty compared to the abstract dialog. This observation coincides with their end-of-study ratings showing that cheating guards with abstract dialog were perceived as easy, although guards with contextual dialog conveyed more concrete information. We need further investigation to confirm this observation.

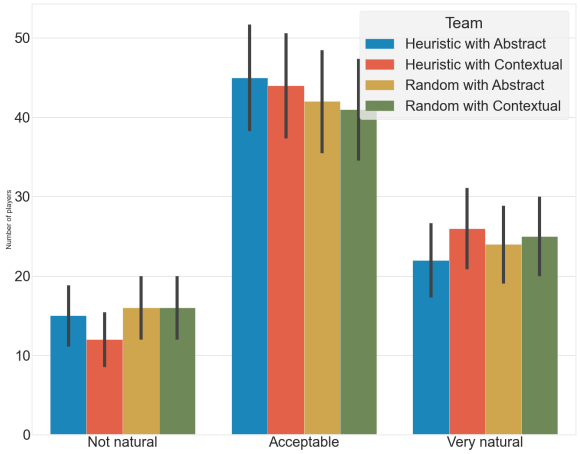
6.5.4 End-of-study Ratings

The participant responses after each round could have facilitated their reflection on the teams they encountered in that round. However, due to the fact that participants did not compete against all teams, there was a considerable variance in their responses. To address this, after



(a) Enjoyment

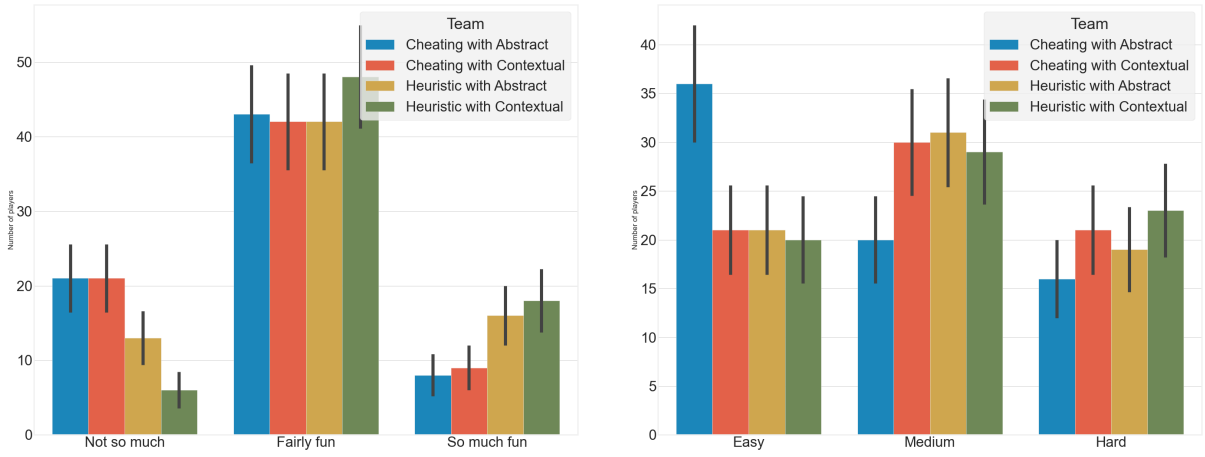
(b) Difficulty



(c) Naturalness

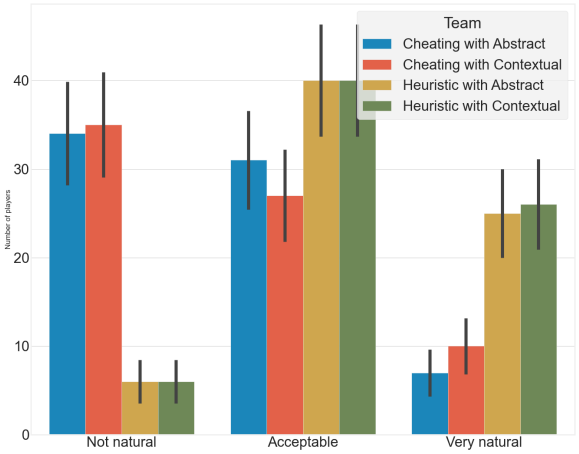
Figure 6.7: The end-of-round ratings for the participants allocated in the "Random" group for the three aspects (enjoyment, difficulty, and naturalness). The error bars represent 95% confidence intervals. [7]

playing against all teams, we requested them to rate the most enjoyable and challenging team they faced. First, we analyze player responses regarding their most enjoyable guard behavior and dialog type.



(a) Enjoyment

(b) Difficulty



(c) Naturalness

Figure 6.8: The end-of-round ratings for the participants allocated in the "Cheating" group for the three aspects (enjoyment, difficulty, and naturalness). The error bars represent 95% confidence intervals. [7]

Enjoyment

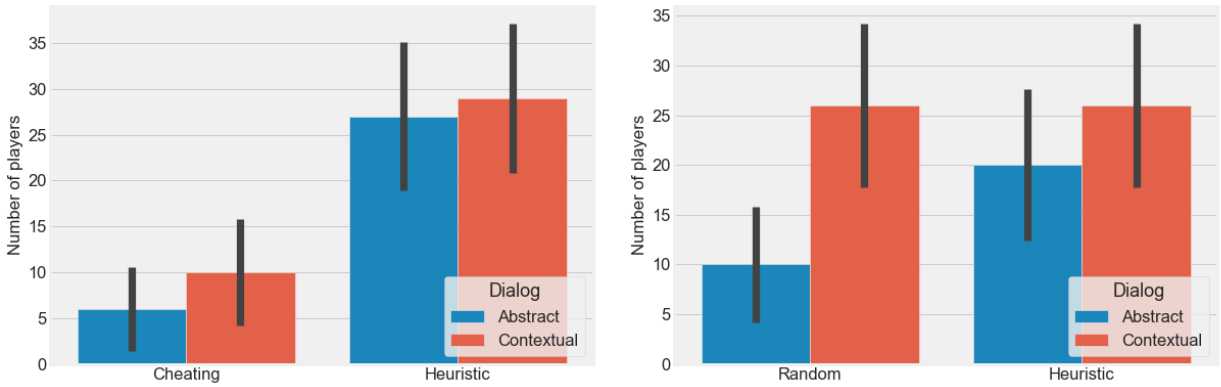
Figure 6.9 presents the ratings provided by both participant groups. Within the "Cheating" group, a majority of players expressed a preference for playing against the heuristic method rather than the cheating guards. We believe this outcome is influenced by the way in which the cheating guards simply converged towards the players, allowing players to exploit the

cyclical nature of the game level to evade the guards effectively. In contrast, the heuristic guards exhibited a greater variety in their search patterns, compelling players to employ diverse strategies. This observation is further corroborated by a Chi-Square goodness-of-fit test, which established that the guard behavior significantly influenced player enjoyment ($\chi^2(1, 72) = [22.22], p < 0.001$). The corresponding values for the Chi-Square goodness-of-fit concerning guard behaviors and dialog types, in relation to player choices of their most enjoyable teams, are detailed in Table 6.7.

As for the impact of dialog type, there was no discernible significant effect on player enjoyment when comparing player ratings of their most enjoyable teams. We suspect that this result stems from players readily distinguishing between the behaviors that had the most substantial impact on their overall experience.

Regarding the participants in the "Random" group, it is evident that the type of dialog significantly influenced their enjoyment, as indicated in Table 6.7. When we compare both guard behaviors with the abstract dialog type, players favor the heuristic method. This observation could be attributed to two potential reasons. First, the heuristic method potentially allowed players to achieve higher scores, leading to a more enjoyable experience. Second, the heuristic method might have exhibited more engaging behavior compared to the random behavior.

However, when the dialog type was switched to contextual, player enjoyment improved to match the rating for the heuristic method. This outcome can be interpreted in several ways, two of which are as follows. First, when combined with contextual dialog, guard decisions became more transparent. This meant that even with random movement, players achieved higher scores by concretely understanding guards' intentions. Which resulted in increasing their enjoyment of that team. The second possible interpretation is that the inclusion of contextual dialog by the guards gave them more lines to announce, which contributed to making them more interesting to players.



(a) "Cheating" behavior group

(b) "Random" behavior group

Figure 6.9: Participants' votes for the most enjoyable teams. [7]

Table 6.7: The chi-square goodness-of-fit test results of the players' most enjoyable behaviors and dialogs. The sample size for comparing the cheating vs. heuristic method is 72, and for the Random vs. heuristic is 82 participants. [7]

	vs Random		**vs Cheating**	
	χ^2	p	χ^2	p
Behavior	1.21	0.26	22.22	<0.001
Dialog	1.21	0.01	0.5	0.47

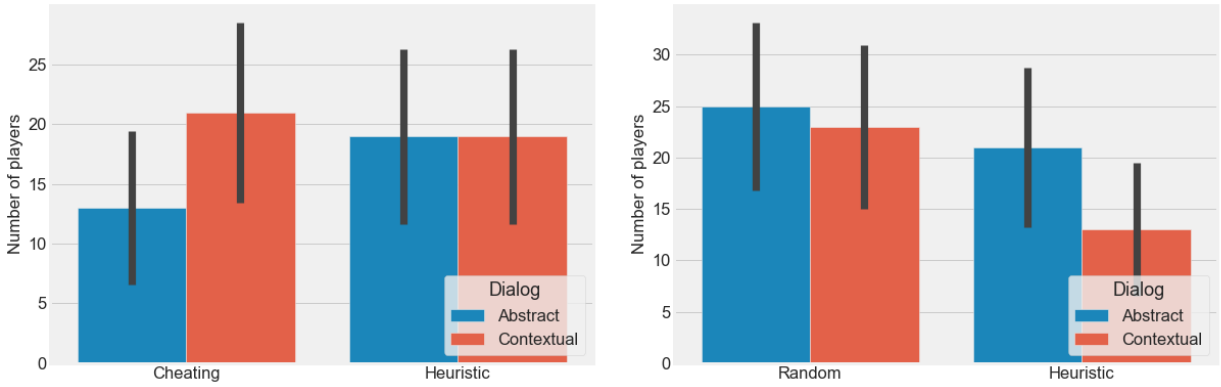
Difficulty

Figure 6.10 presents a bar chart displaying player preferences for the most challenging team they encountered during the study. In the "Cheating" group, although player scores exhibited no significant differences among the various teams, a greater number of players perceived cheating guards with contextual dialog as the most difficult, in comparison to those with abstract dialog. This outcome can be attributed to the fact that guards explicitly announced the name of the room where the player was located, revealing a clear indication that the guards were aware of the player's position. This overt display of knowledge confirmed a sense of an unfair advantage held by the guards, consequently leading players to perceive this team as more challenging. Furthermore, in the heuristic method group, players were

evenly distributed in their selection of the most challenging team, indicating that dialog types did not have an impact on the perceived difficulty.

Comparing the previous result with how players voted for the most challenging team in the "Random" group, we noted that the dialog type influenced the perceived difficulty of the heuristic team. We attribute this variance in player opinions between the two groups to the ease with which players could distinguish between cheating and heuristic behaviors compared to random and heuristic behaviors.

In the "Random" group, the abstract dialog appeared to make the heuristic method seem more challenging to players. This could be explained by the fact that guards were more transparent in their decision-making when employing contextual dialogs, so players could better anticipate their future actions.



(a) "Cheating" behavior group **(b)** "Random" behavior group

Figure 6.10: Participants' votes for the most difficult teams. [7]

To confirm the significance of this result, table 6.8 shows the Chi-Square goodness-of-fit for the guard behaviors and dialogs for player choices of their most enjoyable teams. The results show no statistical significance; however, we believe that further testing with a larger sample size may give more insight.

Table 6.8: The chi-square goodness-of-fit test results of the players' most challenging behaviors and dialogs. The sample size for comparing the cheating vs. heuristic method is 72, and for the Random vs. heuristic is 82 participants. [7]

	vs Random		**vs Cheating**	
	χ^2	p	χ^2	p
Behavior	2.39	0.12	0.22	0.63
Dialog	1.21	0.2	0.88	0.34

6.6 Threats to Validity

Although this study yielded intriguing findings, its validity may be impacted by several limitations. Firstly, the sample size was insufficient to yield statistically significant results; however, observed trends suggest that a larger sample size could offer more conclusive insights.

The primary method for data collection involved participants using an online prototype and completing a multiple-choice questionnaire. The absence of direct observation prevented the capture of qualitative data on player reactions. Moreover, the use of multiple-choice surveys restricted participants from providing additional justifications for their responses.

Concerning the study's duration, we recognize the challenge of assessing whether players could differentiate behaviors like Random and Heuristic after interacting with them briefly. We believe that is essential to create a user study allowing players more extended interactions with the AI, resembling a typical gaming session.

Despite these limitations, the study unveiled suggestive results regarding player enjoyment in the context of different search behaviors and the impact of pairing various dialog types with NPCs. It also points towards future directions for further evaluation and confirmation of the identified trends.

6.7 Summary

Developing sophisticated game AI can be a challenging task, representing a continuous research effort. Intelligent decision-making in games, however, serves more as a mechanism than an ultimate goal and can be substituted with simpler, cost-effective methods when players cannot sense a noticeable difference. Our investigation of a user study within the context of a stealth game revealed that players are capable of differentiating between search behaviors when they are distinctive. This holds true, especially when the behavior appears unjustified based on the local knowledge that NPCs should logically possess. This distinction remains noticeable even if NPCs employ the illusion of intelligence through contextually appropriate dialogs. Conversely, other straightforward approaches like randomization can effectively complement detailed contextual dialogs, resulting in an overall positive gaming experience that rivals or even surpasses that achieved by using more intelligent opponents.

In the upcoming chapter, we shift our focus to the other aspect of the stealth equation. Here, we will be developing a pathfinding method for stealthy NPCs that must navigate through a game environment while avoiding guards with non-deterministic patrol patterns.

Chapter 7

Stealthy PathFinding

In the previous chapters, we focused on creating a dynamic behavior for the guards to patrol or search for an intruder. This chapter focuses on creating intruder behavior against adversaries with non-deterministic patrol behaviors.

In many action and role-playing games, achieving stealth-oriented or discreet pathfinding stands as a common requisite. This involves players traversing game levels while ensuring their invisibility to adversarial agents. While this responsibility mostly lies with the player, the pursuit of computational solutions holds significance within the domains of game design, testing, and the enhancement of immersive interactions involving stealthy allies or other NPCs, necessitating an algorithmic approach.

When the movement of adversaries adheres to a predictable design, the issue can be simplified into dynamic pathfinding, a realm where various heuristic solutions have been recommended in prior research [119]. However, in a multitude of gaming situations, the opponents might exhibit non-deterministic behavior, such as investigating a sound or incorporating an element of randomness into their patrol patterns. In principle, predicting adversary routes can be beneficial to stealthy pathfinding; however, recent methods lean towards a conservative stance, predominantly evaluating the probabilistic trajectories of adversaries through past training data, all the while overlooking the spatial characteristics inherent to the game level [57].

Within this chapter, we present a proposition for achieving stealth-oriented pathfinding that relies on a derived level roadmap constructed from the game environment itself, thus reducing the necessity of leaning on prior observations of adversary actions. This roadmap assumes the role of abstraction, simplifying predictive modeling of adversary movements and more precise allocation of the probabilities of their future positions. Our proposed strategy's effectiveness is highlighted through empirical examination, showcasing its capacity to enhance pathfinding in scenarios characterized by non-deterministic stealth challenges. Additionally, we explore diverse parameters intrinsic to our approach, which impacts its efficacy. This exploration shed light on how various elements can be used to modulate the level of difficulty posed by the stealth scenarios.

The specific contributions of this work are as follows:

- We present a new approach for moving stealthily within a defined area against adversaries whose patrol patterns are non-deterministic. This method addresses a considerable limitation in existing solutions that usually assume guards have fixed patrol behaviors.

- We conduct empirical experiments to evaluate the method's efficacy across multiple maps from available commercial games. This evaluation involves comparing our results to baseline techniques, revealing the influence of variables on performance beyond the factors of the adversary's speed and vision. These additional factors include the count of adversaries, map characteristics, and the initial spawn position.

The structure of this chapter is as follows. Initially, Section 7.1 presents the standard scenario employed to evaluate the effectiveness of the stealthy pathfinding method. Subsequently, a thorough explanation of our method is provided in Section 7.2. Following that, in Sections 7.3 and 7.4, we outline the experimental setup, encompassing the parameterization of the method and a performance comparison of the stealthy pathfinding approach. Finally, in Section 7.5, we present the results obtained from our experiments, including covert performance metrics and the computational resources utilized by the method in our system.

7.1 Scenario

There are numerous prospects for covert motion planning, each with significant distinctions based on factors such as whether guards move in a deterministic manner, the properties of their FOV, and whether the intruder possesses complete or partial information of the surroundings and guard locations. In this study, we reuse the prototype we developed in previous chapters, which models game levels as 2D polygons with holes (representing obstacles) viewed from a top-down perspective. Figure 7.1 shows a screenshot of the game, and the elements are:

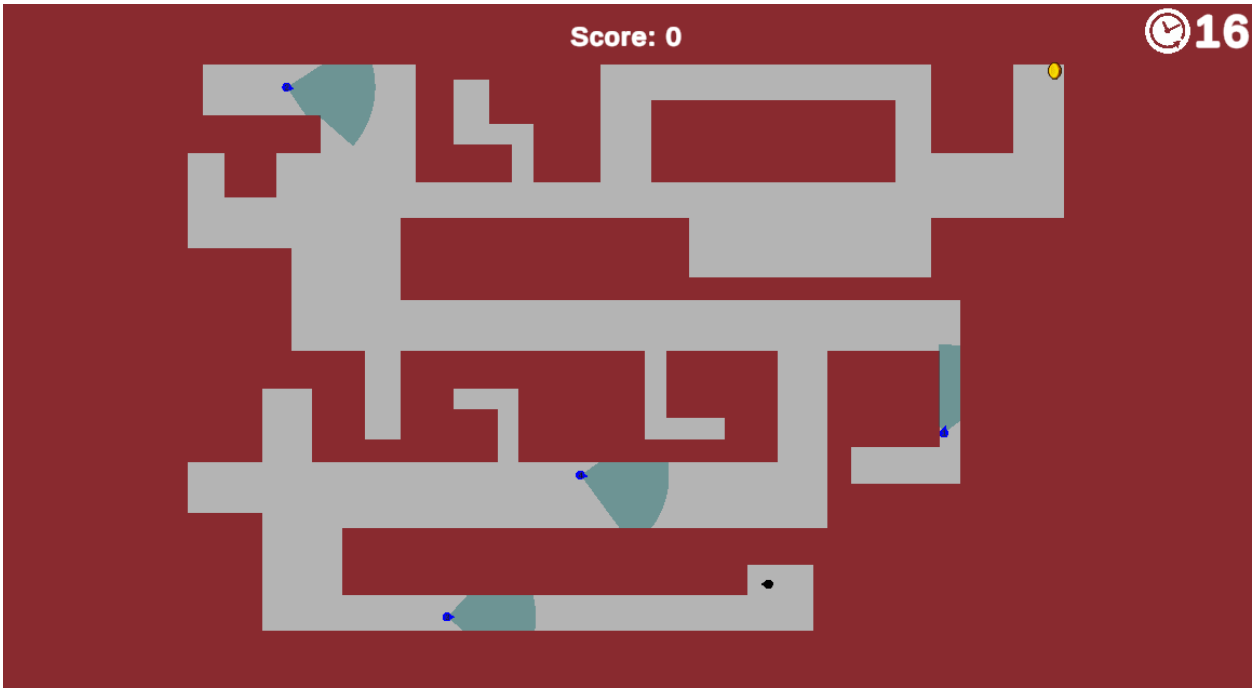


Figure 7.1: A screenshot of our prototype. The intruder (black dot) is tasked with reaching the coin (yellow dot) without being discovered by the guards (blue dots).

- Game level: The game level is a polygon with holes. The walkable area is colored grey, and the unwalkable area is red.
- Guards: At the game's start, a predetermined number of guards are distributed across random locations. The initial guard is positioned near a randomly chosen vertex of the outer polygon. Subsequently, each additional guard is placed at the furthest vertex,

determined by path distance from the previously positioned guards. This procedure is iterated until all guards are appropriately positioned. It is important to note that the guard movements are not deterministic, and each guard has a limited FOV with a fixed radius and range.

- The intruder: The central aim of the intruder is to reach the destination while evading detection by the guards. We position the intruder's starting point at the corner furthest from the coin and the guards. In our study, we equip the intruder with comprehensive knowledge of the guard positions, velocities, and orientations, to improve decision-making. Moreover, we provide the intruder with complete information about the game level, eliminating the need for exploration and enabling them to focus exclusively on the intricate task of stealthy path planning.

- Coin: A coin appears at the most distant corner from the intruder, serving as the primary destination for their objective.

- Time: The intruder has a predetermined time window to achieve their goal. Specifically, we set the scenario time to 120 seconds. The scenario concludes with either a successful outcome, denoted by the intruder reaching the coin within the given timeframe, or a failure if they are unable to reach it in time or are detected by the guards.

7.2 Methodology

This section outlines our approach to planning paths that avoid detection from guards with non-deterministic movements. As a general strategy, the intruder utilizes the positions and velocities of the guards to make predictions about their future trajectories. Based on these predictions, the intruder devises a secure path that bypasses crossing the guards' paths while still reaching the primary goal. Nevertheless, if no such path is available, the intruder seeks out reachable locations on the map. These locations may either bring them closer to the

primary destination or offer a safer position if there is a risk of being spotted by a guard. Algorithm 7 provides a high-level pseudo-code version of our method.

Firstly, we use our knowledge of the current location and velocity of the set of guards G to predict their potential trajectories on the roadmap N (Line 2). An abstract representation of the game level mainly guides these predictions, as described in section 7.2.1. By modeling the guards' motion, we can estimate the "risk" of the current intruder's location i, which we explain in section 7.2.2 (Line 3). Depending on this value, if the intruder is in a safe spot, it tries to get closer to the goal destination by checking if there are any safe paths to the goal or other locations within the game level. We depict the path construction process in section 7.2.3. If there is no safe path to the goal, we search for sub-goal locations that can heuristically bring the intruder closer. We specify how we identify these sub-goals in section 7.2.4 (line 6). In contrast, if the intruder is at risk of being spotted, it focuses on finding a safer spot. Once a safe and reachable location is identified, we follow it as our path, continuously evaluating its safety and discarding it if it fails, as detailed in section 7.2.5. In such a scenario, we recalculate a new path to the goal or a sub-goal using the same approach.

The intruder i uses a roadmap defined as a set of nodes N' to plan and avoid the patrolling guards G to reach a goal destination d. This method uses a list of dynamically generated hiding spots (H), with H' being a selected subset of H; section 7.2.3 shows the selection process. r is the risk value assigned to the intruder's current location.

7.2.1 Modelling Guard Motion

Our method's initial phase involves predicting the future locations of the guards. To accomplish this, we begin by constructing a simplified abstraction or roadmap of the game level. We then utilize this abstraction, along with the current positions and velocities of the guards, to generate potential trajectories of their movement. By incorporating the roadmap structure, we limit and guide the range of potential guard trajectories.

Algorithm 7 Stealthy Path Planning Algorithm

Require: H, the set of hiding spots.
Require: G, the set of guards.
Require: N, the roadmap.
Require: i, the intruder.
Require: d, the main goal.

```
1: function StealthyPathFinding(H,G,N,i,d)
2:     N' = ModifyRoadMap(G,N)          ▷ By using the guard data, modify the roadmap.
3:     r = GetCurrentRisk(N',i)
4:     SetPathDestination(N',i,r,d)                ▷ Try to find a safe path to main goal
5:     if not i.has_Path then                      ▷ Check if there is a path to the main goal
6:         H' = GetHidingSpots(N',H)                ▷ Get a list of possible hiding spots
7:     end if
8:     while not i.has_Path and H' not empty do
9:         AssessHidingSpots(H')          ▷ Update the heuristic values of the hiding spots
10:        h' = GetBestHidingSpot(H')
11:        SetShortestSafePath(i,h')
12:        if not i.has_Path then
13:            Remove(h',H')
14:        end if
15:    end while
16:    if not PathSafe(i,r) then
17:        CancelPath(i)
18:    end if
19: end function
```

Representing the Environment

Since guards have non-deterministic motion, they may change direction arbitrarily while patrolling or moving through a game level. However, assuming human-like behavior, they tend to adhere to the overall level geometry. To take advantage of this expectation, we require an abstract representation of the game level that captures its overall geometry. As a roadmap, we again utilize the straight skeleton graph to accomplish this because it captures the main features of a polygon, especially in the more "maze-like" environments typical of stealth games [45]. Since the goal of using the roadmap is to provide a general representation of the possible paths guards may take through the game level, we removed "extra" edges that connected the graph to corners, as illustrated by the yellow lines in figure 7.2. Importantly, this graph is only used for the intruder's estimation and planning, and the intruder's motion

utilizes the game's standard pathfinding mechanism NavMesh, which is not constrained by the roadmap. Moreover, this graph can be created just once, and the resulting data structure can be stored in a file, thus diminishing the computational time required for subsequent executions on the identical map.

Marking Guard Trajectories

In order to indicate potential future positions of a guard, we initiate the process by projecting their location onto the nearest point along the roadmap. This projection serves to approximate their current position within the game level. Subsequently, we can extend the projected point along the roadmap in alignment with the guard's velocity, effectively enabling the assignment of likelihoods regarding the guard's potential presence at specific locations along the roadmap's nodes. It's important to note that our trajectory plotting hinges on the presumption that guards adopt exploratory movement patterns to prevent retracing their steps. Given that the Euclidean distance between nodes in the roadmap might be large, we introduce interim nodes temporarily to enable a more detailed estimation. These trajectories take the form of simple geometric projections onto the roadmap, highlighted in red in Figure 7.2. Algorithm 8 describes how the propagation is done for a guard. The total propagation distance is limited depending on the guards' speed and their FOV radius.

The approach described in algorithm 8 involves identifying the edge of the roadmap to which the guard's position has been mapped and incorporating a starting node l into the graph at that location. Assuming l corresponds to the roadmap edge (p, n) (where p and n represent nodes in the roadmap), and n is in the direction of the guard's forward velocity, we insert temporary nodes into the graph at fixed intervals of s up to a total distance of d from the starting point l. d is the projection distance to reflect the guard's trajectory. It is crucial to choose a value that guarantees a representative distance of the guard's future motion.

Along the propagated trajectory, the inserted nodes in the roadmap represent possible future positions. Each node is also associated with a numerical value that indicates the

Algorithm 8 Adding the possible trajectories of a guard on the roadmap.

```
1: function AddGuardNode(p,l,n,d,s) ▷ Add a node at a step s from position l on the edge p to n
2:     if d > s then ▷ Make sure the step size is less than the remaining distance
3:         d' = d − s
4:         s' = s
5:     else
6:         d' = 0
7:         s' = d
8:     end if
9:     if Distance(l,n) > s' then ▷ Place a node if the step is less than the Euclidian distance to the next node.
10:        n' = InsertNodeOnEdge(p,l,n,s')
11:        AddGuardNode(n',n',n,d',s) ▷ Recursively place another node on the same edge n' to n
12:    else
13:        for each c in n.connections do
14:            if c is not p then
15:                AddGuardNode(n,n,c,d',s) ▷ Recursively place another node on the connecting edge n to c
16:            end if
17:        end for
18:    end if
19: end function
```

likelihood of the guard observing that future point, which we call the "risk" value. Figure 7.2 shows an example of how the trajectories of the guard are modeled in our system.

Risk values are heuristic, range between 0 (safe) and 1 (currently in the guard's FOV), and are otherwise scaled according to the path distance of the node from the guard's starting position. To calculate the risk value for a node n and a guard g, we use the equation

$$r(n) = 1 - \frac{Distance(n, g)}{g.FovRadius * (g.Speed + 1)} \tag{7.1}$$

where $Distance(n, g)$ is the path distance between node n and guard g. The denominator is the normalizing factor, which is the projection distance d in algorithm 8. In the case of overlapping FOV from multiple guards, the maximum risk value is considered.

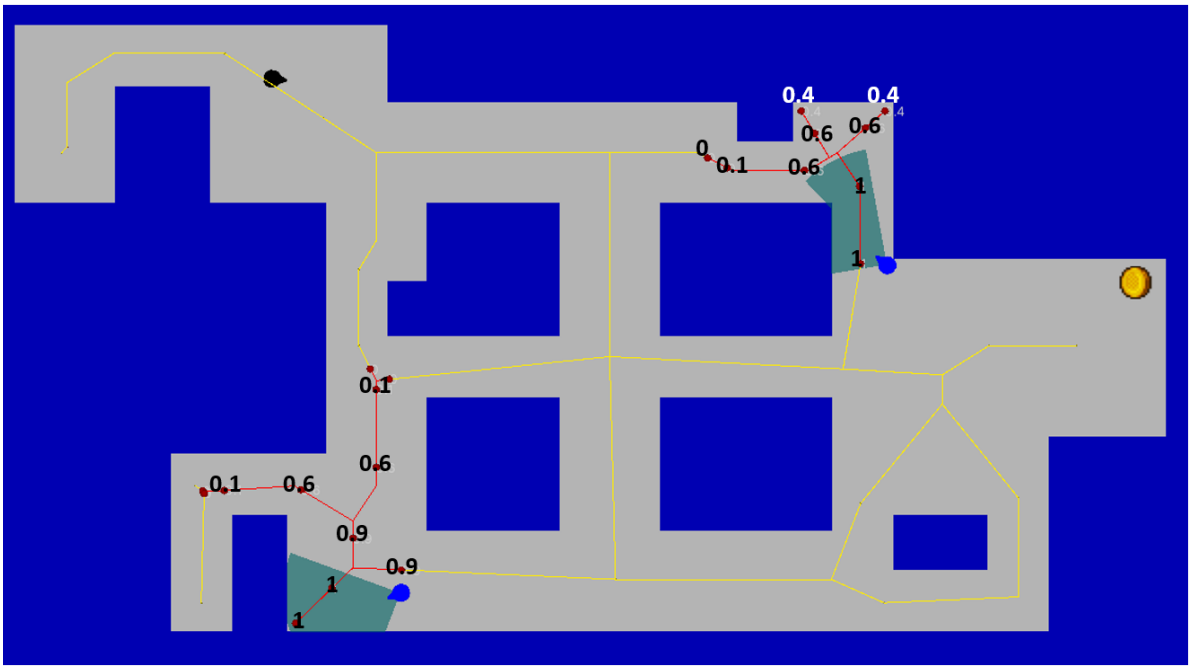


Figure 7.2: An example of how trajectories are laid out on the roadmap. Guards are in blue, the roadmap is in yellow, and the red line segments represent possible guard positions propagated along the graph. Each red node represents a position along the roadmap with an associated risk value, shown in black and white text.

7.2.2 Define the Intruder's Risk

Once we have marked the potential future positions of the guards on the roadmap, our next step is to strategize the intruder's next move. Typically, the intruder wants to reach the goal destination by avoiding the guard trajectories. However, sometimes guards might be in positions that pose a larger threat to the intruder's current position. One strategy is to avoid the guards and relocate to a more secure spot. To do this, we need to evaluate the level of risk associated with the intruder's present position before selecting an appropriate action. If the intruder's current position is risky, the top priority is to relocate to a safer location. On the other hand, if the intruder is currently in a secure location, then identifying a safe path toward the goal takes precedence.

Calculating the Risk of a Position

To assess the level of risk, we consider the risk value of the node that has the highest risk within a specific range from the intruder's current position. This range corresponds to the radius of the guard's FOV. By using this approach, we can obtain an upper limit for approximating the risk value of being imminently detected.

The risk associated with a particular position aids in determining the intruder's situation. When the risk value is below a given threshold, it implies that the intruder is secure. However, if the risk value exceeds the threshold, it indicates that the intruder is at risk of being detected. Consequently, the intruder selects an appropriate action based on their safety status.

7.2.3 Set the Path to Destination

The intruder devises a plan for movement, which hinges on three potential scenarios. These scenarios depend on the intruder's current safety status and whether they can identify a path to their destination. These scenarios are: safe with a clear path to the goal, safe without a clear path to the goal, and unsafe. As outlined in section 7.2.2, the intruder's safety status is determined by comparing their risk value to the risk threshold.

When the intruder is safe, it prioritizes planning a safe path to the goal. The first step involves utilizing A* on the roadmap to locate a feasible path to the goal. To enhance the chances of generating a safe path, nodes on the graph with a risk value higher than a predetermined threshold are omitted. This measure improves the likelihood of avoiding detection by the guards and navigating through secure locations in the game level. Figure 7.3 provides an instance of a path obtained after removing these segments. In section 7.2.3, we examine the effect of varying the threshold levels on the performance.

If the intruder fails to find a safe route to the goal, it can move towards a series of procedurally generated positions within the walkable space. These positions are known as hiding spots, and a comprehensive explanation of how they are determined is presented

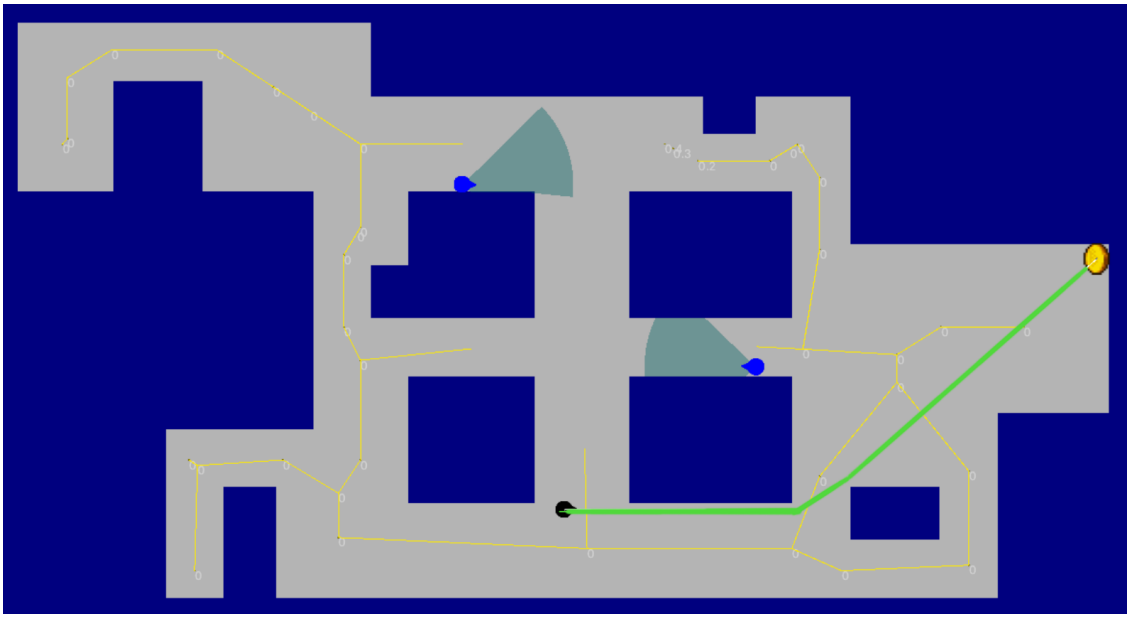


Figure 7.3: Once nodes with high-risk values have been eliminated, the roadmap is temporarily reduced. A* algorithm is then utilized to generate a secure path for the intruder to reach the coin. The optimized path, denoted by the green line, is obtained using the NavMesh.

in section 7.2.4. Given a range of possible hiding spots, the task is to identify a subset of feasible spots as sub-goals. To accomplish this, we select spots situated near the dead-ends that arose during the failed A* path search. Hiding spots will surround each of these regions, and each hiding spot can serve as an intermediate goal, bringing the intruder closer to their ultimate destination.

To minimize the computation time, we limit the scope of potential hiding spots to a nearby "neighborhood" surrounding the dead-ends. This area comprises a set of spots with an unobstructed line of sight between them. Figure 7.4 illustrates an example of this scenario.

After obtaining a set of potential hiding spots, we employ two primary factors to determine the optimal option. These factors are the risk value and the path distance to the goal. The position with the least risk and the shortest distance to the goal will be favored. Moreover, the degree of emphasis placed on these two criteria can be modified to give preference to either more conservative or daring choices.

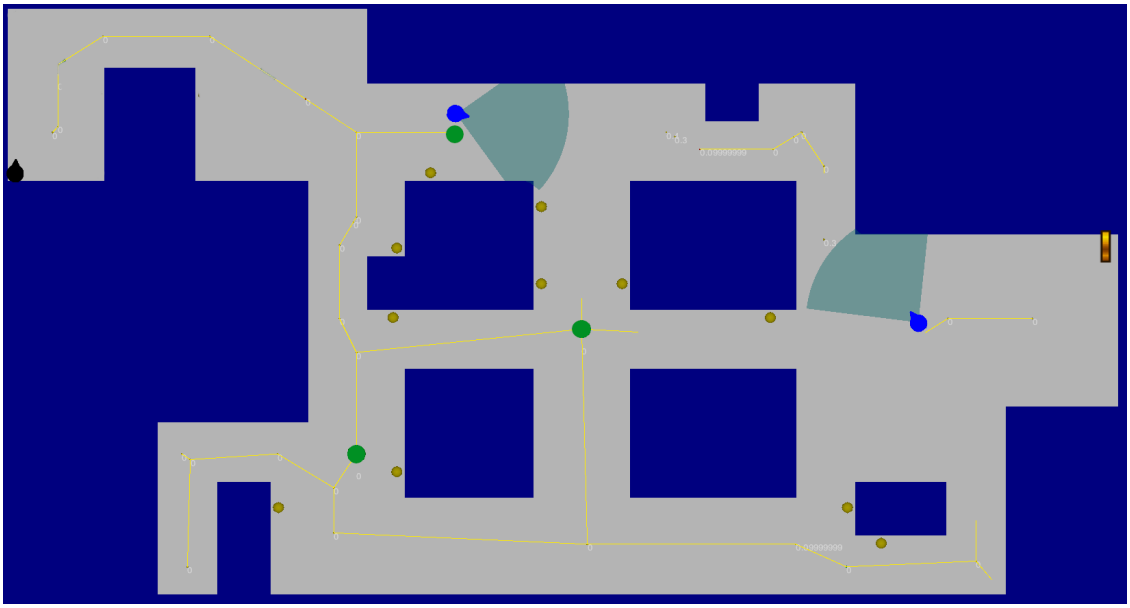


Figure 7.4: The green circles represent the search limits encountered during the A* search, while the smaller yellow circles depict the possible hiding spots that the intruder can travel towards.

On the other hand, **if the intruder is no longer safe, it gives precedence to relocating to a secure location instead of advancing towards the goal**. As avoiding detection is paramount, the intruder prioritizes identifying nearby hiding spots and utilizes Dijkstra's search algorithm with a fixed search radius on the roadmap instead of A*. We switch the search algorithm here to find all paths from a point to have more options. The neighborhood encompassing the closest hiding spot to each terminal node in the search is included in the set of potential hiding spots. Subsequently, we can determine the safest hiding spot using the lowest risk as the primary criterion and the highest fitness for hiding as the secondary criterion. In section 7.2.4, we provide details of various strategies that can be used to prioritize the potential hiding spots.

7.2.4 Hiding Spots Placement

The intruder can use these intermediate spots to move undetected. Although hiding spots could be located anywhere less likely to be observed, we have chosen a specific subset for

efficiency. Our design places these spots procedurally at the corners of the game level, with one spot in a convex corner and two on either side of a reflex corner. Figure 7.5 shows the process for generating these spots. To determine the ranking of each spot, we calculate its utilities using equation 7.2, with the main criterion being the level of risk and fitness as a secondary factor.

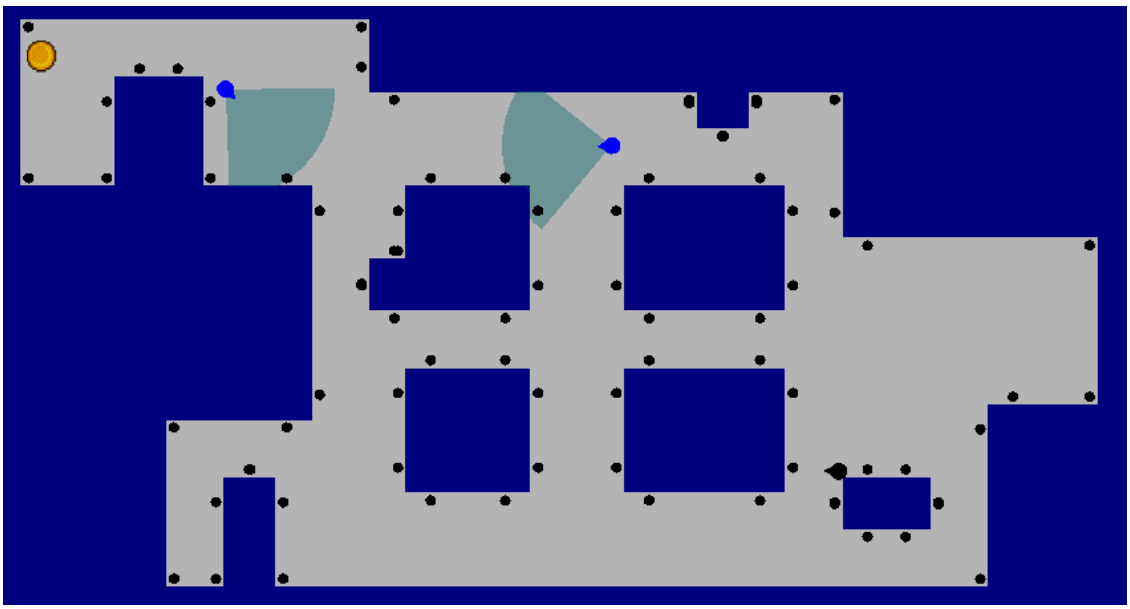


Figure 7.5: Black dots indicate the placement of the hiding spots, which are situated at the corners of the walkable area. In the case of two spots within a threshold distance, we removed them and placed one spot instead in the midpoint. This step is done to reduce redundancy among hiding spots.

Utilities

To determine the fitness of a hiding spot, we consider four spot properties: goal utility, guard proximity utility, occlusion utility, and cover utility. A range of numerical values from 0 to 1 was assigned to each hiding spot, which includes:

- **Risk Utility:** This is the value of the highest risk value of a node within a distance that equals the radius of the FOV of the guard. This value reflects the likelihood of the guard detecting that spot, so the higher it is, the riskier the spot is considered.

- **Goal Utility:** The normalized path distance from the hiding spot to the goal is calculated by dividing the actual path distance by the game level diameter to normalize the result. This value indicates the distance between the spot and the goal.

- **Proximity Utility:** The distance from the hiding spot to the nearest guard is calculated and normalized by dividing it by the game level diameter. The higher this value, the further away the spot is from the closest guard.

- **Occlusion Utility:** This value represents the number of guards with a direct line of sight to the hiding spot, normalized by dividing the number of such guards by the total number of guards in the game level. A 0 indicates that no guards have a line of sight to the spot, while a 1 implies that all guards do.

- **Cover Utility:** This value reflects the level of concealment of the hiding spot from the walkable area in the game level. It is computed by dividing the area of the visibility polygon from the spot by the total area of the walkable space. We then subtract the result from 1. Thus, a higher value indicates a more hidden spot.

Equation 7.2 illustrates how the fitness of a spot is calculated.

$$\begin{aligned} f(x) = w_g * (1 - \mathit{Goal}(x)) + w_p * \mathit{min}_{g \epsilon G}(\mathit{Proximity}(g, x)) + \\ w_o * (1 - \mathit{Occlusion}(x)) + w_c * (1 - \mathit{Cover}(x)) \end{aligned} \tag{7.2}$$

In equation 7.2, x represents a hiding spot, and w_g, w_o, w_c, and w_p are fixed weights that range from 0 to 1, used to adjust the goal, occlusion, cover, and proximity utility values. The higher the computed fitness level, the more suitable the spot is for hiding. Once the optimal spot is identified, the intruder plans a safe path to it. To simplify the design and accommodate the possibility that different situations require different weights, we defined two settings of these weights; the first is when the intruder is safe, and the second is when it is considered unsafe. We explore the impact of these weights in section 7.3.

7.2.5 Ensuring Path Safety

Since the intruder does not know the guard's plans, it needs constant confirmation of the path's safety. Once that path is compromised, we discard it and request a new path. First, we identify the riskiest intersection points of the guards' paths. We project the guards' predicted trajectories on the intruder's route. If a projected point is within the FOV of that guard it is considered as a possible interception point. At each interception point, we identify its risk value. We do that similarly to how we calculated it for the intruder's position in section 7.2.2. Furthermore, we focus only on the highest risk point. Figure 7.6 shows an example of an interception point projected on the intruder's planned route.

Once we have determined the most unsafe interception point, we establish two primary variations to determine whether a path is too dangerous to pursue and should be disregarded:

- **Simple Risk Comparison:** The primary objective of this approach is to prevent the intruder from being placed in a more unsafe location than their current one. We perform a straightforward comparison of risk values. If the maximum risk associated with an interception point is lower than the intruder's current risk value, the path is deemed safe for the intruder to pursue.

- **Distance Calculation:** In this step, the intruder estimates the time required to reach the potential interception point and compares it to the time it would take for the associated guard to detect the point. If the guard's FOV reaches the interception point before the intruder, the path is considered dangerous and is eliminated.

This check is done at every time step; however, it can be executed every n frames to reduce the required computation. Next, we must determine the values to be assigned to the weights in our method.

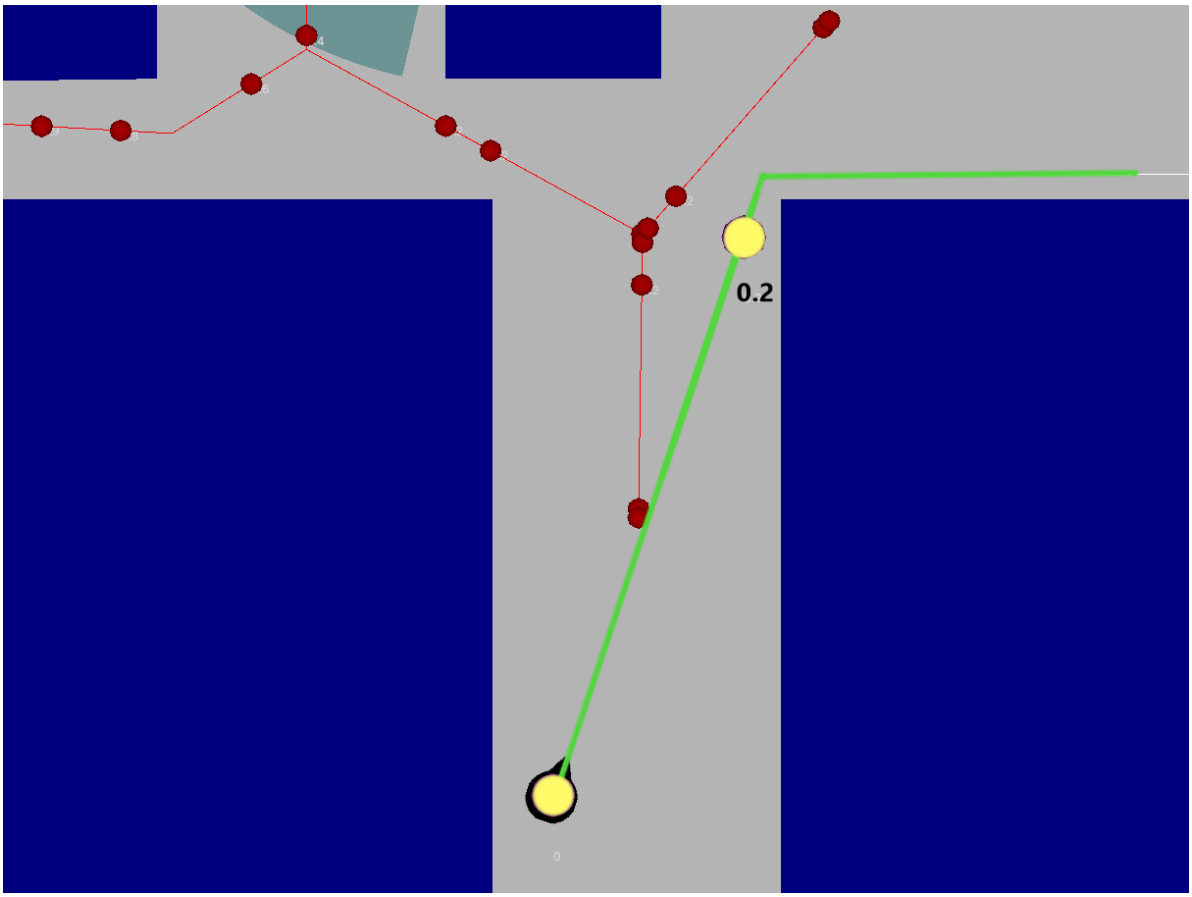


Figure 7.6: The green line represents a portion of the intruder's intended path. Based on the guard's predicted trajectory, we anticipate possible interception points along the intruder's path, shown in yellow. These points are assigned a non-zero risk value that indicates the probability of the guard spotting the intruder at that location. Even though there are several possible interception points, we only consider the possible interception point with the maximum risk value per guard, which is shown in yellow with a risk value of 0.2.

7.3 Weight Tuning

Our method contains various parameters that can influence its performance. To evaluate it, we must initially establish the parameter values that maximize the intruder's success rate in reaching a goal undetected. These parameters include:

1. **Safety Threshold:** This value plays a role in determining when the intruder is classified as safe or not. When the risk value surpasses the safety threshold, the intruder is regarded as being in an unsafe situation. This threshold serves as a means to fine-tune

the level of caution the intruder should exercise in assessing specific scenarios as risky or not. The value for this threshold ranges from 0 to 1, with 0 indicating the highest degree of prudence and 1 representing the lowest. In our investigation, we examine two distinct values for this threshold: 0 and 0.5.

2. **Utility Weights:** If the intruder encounters a situation where an immediate safe path to the goal is unavailable, it will seek out a temporary hiding spot as a strategic move to subsequently approach the goal. As clarified in section 7.2.4, the choice of the hiding spot relies on an overall utility calculation. Each weight within this calculation carries a value that ranges from 0 to 1, signifying its influence on the overall utility. We outlined two distinct weight setups: one applicable when the intruder is in a safe state and another when it is deemed unsafe. These dual settings allow the intruder to prioritize hiding spots based on their safety condition. For each weight in equation 7.2, we've considered two values, simplifying the possibilities to 0 or 1. Consequently, this results in eight variables, each possessing two potential values.

In order to evaluate the most effective combinations of these weight settings, we conducted a series of 15 rounds for each combination. We then compared the success rate of the intruder reaching the goal destination within a time limit of 120 seconds. To simplify testing, we selected a single map, "Among Us", due to its well-balanced mix of maze-like structures and open areas.

Following the execution of all possible combinations, our findings indicated that a more cautious approach by the intruder resulted in an improved success rate, as illustrated in Figure 7.7. This figure presents a bar chart depicting the influence of the risk threshold on the success rate.

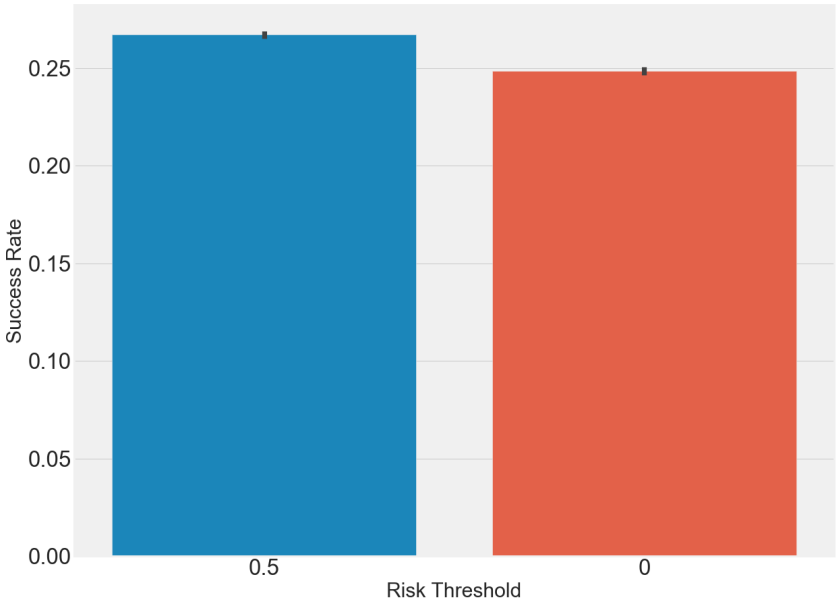
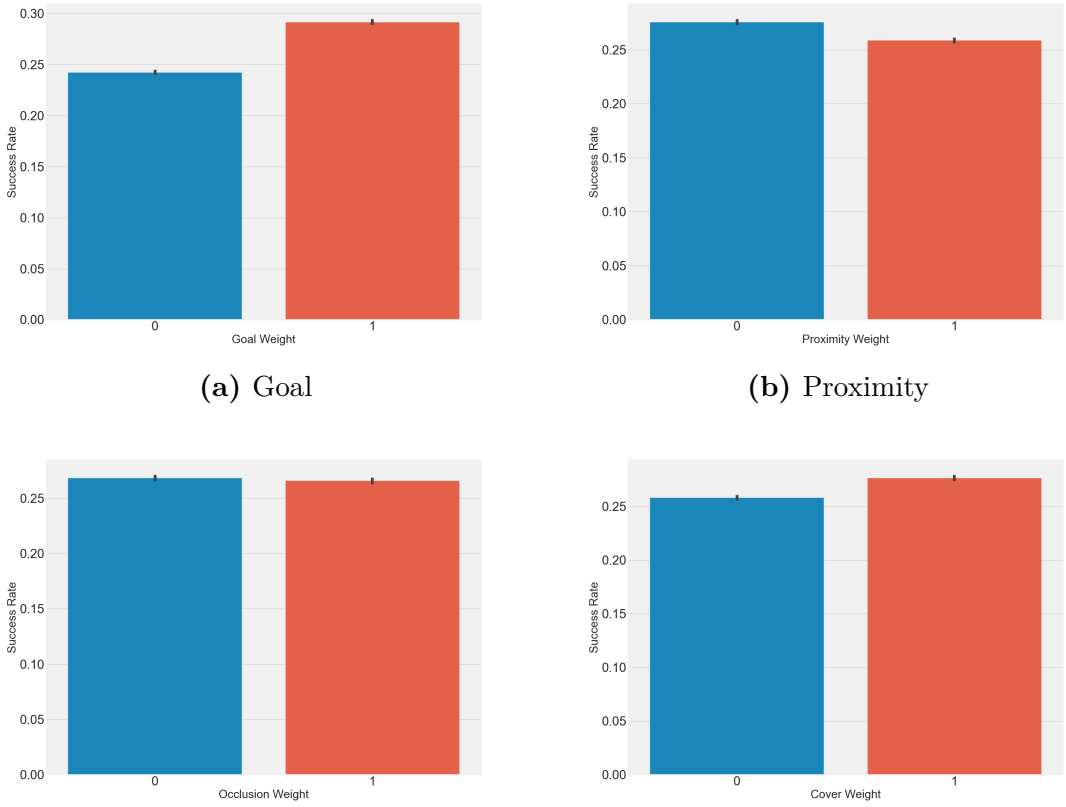


Figure 7.7: Success rates categorized by the weights assigned to the safety threshold. The error bars denote 95% confidence intervals.

Concerning the weights applied when the intruder is in a safe state, Figure 7.8 illustrates the influence of each weight on the success rate. Predictably, opting for hiding spots in closer proximity to the goal led to a higher rate of success, as seen in Figure 7.8.a. Furthermore, selecting well-concealed hiding spots also positively impacted the success rate. However, intriguingly, neglecting the presence of guards near a hiding spot appeared to enhance the success rate.

Regarding the weights used when the intruder is in an unsafe condition, Figure 7.9 demonstrates an interesting observation: the goal weight exhibited no influence on success. This observation aligns with logic, as in such circumstances, the intruder prioritizes securing a safe hiding spot, irrespective of its proximity to the goal, to avert potential failure.

Based on our analysis outcomes, we established the following parameters: we configured the risk threshold to 0.5. When the intruder is in a safe state, we assign weight values of 1, 0, 0, and 0 to the goal, proximity, occlusion, and cover, respectively. Conversely, when the intruder is considered unsafe, the corresponding weights were adjusted to 0, 0, 0, and 1.

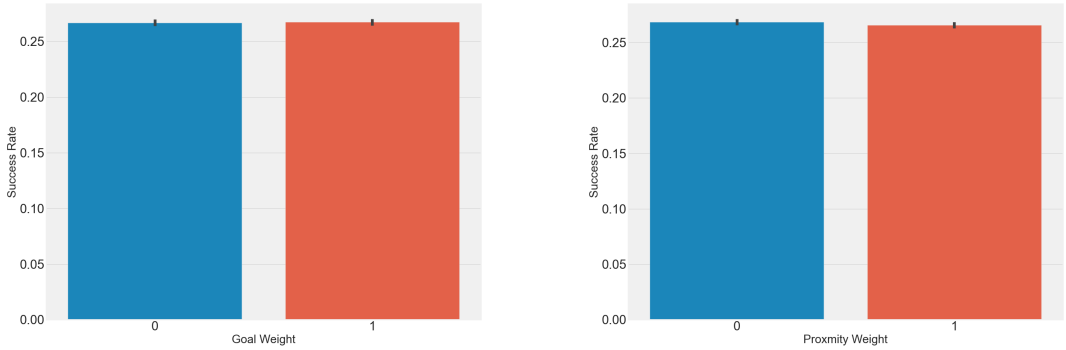


(a) Goal **(b)** Proximity

(c) Occlusion **(d)** Cover

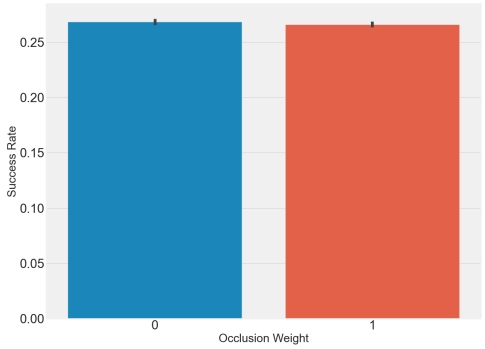
Figure 7.8: Success rates by weights when the intruder was safe.

Discovering the optimal weight configuration for our methods constitutes an intriguing direction for further research. We believe that more efficient approaches, such as using Reinforcement Learning techniques, could potentially bring about significant enhancements to our method [111]. This could facilitate the creation of more dynamic weight settings, defined based on various inputs, including factors like the layout of the game level or the number of patrolling guards. However, within the scope of this thesis, our primary objective is to introduce and implement this method. In the following section, we will provide a description of the experimental setup employed to assess our method.

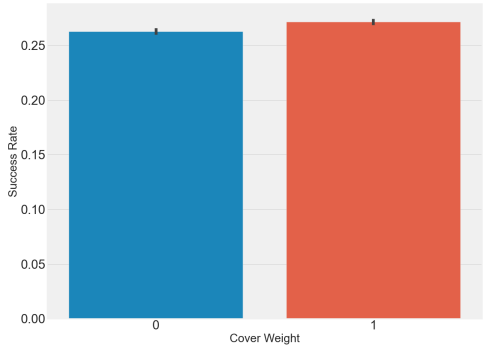


(a) Goal

(b) Proximity



(c) Occlusion



(d) Cover

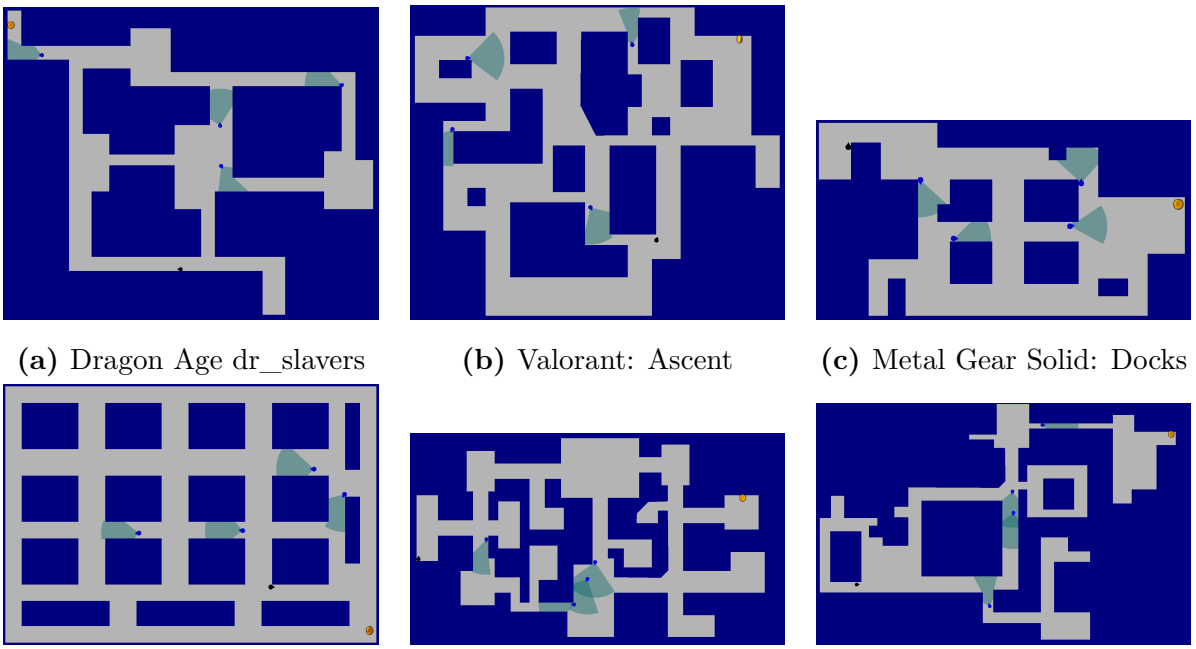
Figure 7.9: Success rates based on weight configurations when the intruder was in an unsafe state.

7.4 Experiment

This section outlines the elements we have established for our experiments to test this methodology. We will conduct 200 rounds of testing, each consisting of a game-level map, an intruder, and a group of guards assigned to patrol the walkable area. The primary objective of the intruder is to navigate to a specific target location while evading detection from any of the guards, all within a fixed time limit. For each round, we define a map, intruder behavior, and guard parameters.

7.4.1 Game-level Maps

Similar to our experiments in this thesis, we incorporated numerous maps that drew inspiration from commercial video games and benchmark maps available in movingAI [106]. For reference, these maps are depicted in figure 7.10.



(a) Dragon Age dr_slavers **(b)** Valorant: Ascent **(c)** Metal Gear Solid: Docks

(d) Warehouse **(e)** Among Us **(f)** Alien Isolation

Figure 7.10: The game-level maps we employed in our experiments: dr_slavers from Dragon Age, a vectorized rendition of the map available in movingAI; Ascent from Valorant; Metal Gear Solid: Docks, a map that has been previously used in stealth pathing analysis [119]; Warehouse, which represents a location with high occlusion but multiple path options; and Among Us and Alien Isolation, adapted from the corresponding video games.

7.4.2 Intruder's Behavior

In order to evaluate our approach, we contrasted its performance against that of two simplistic baseline behaviors for an intruder:

1. **Shortest Path:** The intruder uses the NavMesh algorithm to determine the shortest path towards the objective without considering the presence of guards. This behavior

represents the quickest feasible solution for the intruder and therefore serves as the upper boundary baseline.

2. **Simple:** The intruder waits for a random interval ranging between 5 to 20 seconds. It then identifies the farthest hiding place from all the guards and takes the shortest path toward it, regardless of safety concerns. The primary objective of this behavior is to remain concealed, with less emphasis placed on reaching the goal. This behavior represents the other side of the baseline, where the intruder favors safety more.

7.4.3 Guard Parameters

As for the guards, we consider several parameters that can affect this scenario:

Patrol Behavior

The ability to reach a location undetected is largely influenced by how well the guards perform their patrol duties. We examine three primary guard patrol strategies to investigate the impact of different patrol behaviors on the intruder's objective.

1. **Random:** This represents a basic approach to guard behavior. In this method, each guard autonomously selects a random destination to reach. After reaching the designated spot, they then pick a new one and repeat the process. This generates a dynamic patrol behavior that achieves reasonable map coverage. However, it still provides a non-trivial challenge since it is hard to predict due to the lack of an observable pattern.

2. **RoadMap:** We explained this method in section 3.3. Our findings in section 3.6.1 indicate that this behavior resulted in superior coverage than Random, as it prioritized covering the critical areas of the game level. Moreover, in section 4.4, it was noted that this approach was the most appealing to human players.

3. **VisMesh:** In section 3.4, we provided an explanation of this approach. Our analysis in section 3.6.1 suggests that this behavior resulted in the highest degree of coverage

compared to the other patrol methods. Consequently, as described in section 4.4, it was the most challenging for human players to navigate.

Guard Placement

The initial arrangement of guards concerning the intruder (as well as each other) may influence the outcome of the experiment. Consequently, we outline three primary styles of guard placement.

1. **Random:** We randomly place each guard on the map, which may result in some guards being located close to one another.

2. **Separation:** By spreading the guards, we may reduce their overlap and thus provide a good start position for the guards to patrol. To place the guards, we start by randomly situating the first guard on the map. Next, we sample a predetermined number of random positions and select the location farthest away from the previously placed guard. We repeat this process for each subsequent guard.

3. **Goal Position:** To make reaching the goal more difficult, we can situate all guards at the goal position.

7.5 Results

To assess performance, we conducted 200 rounds for every combination of intruder behavior, guard placement, team size, patrol strategy, and map. Each iteration was timed at a constant duration of 120 seconds. The evaluation of intruder performance is categorized primarily through a key metric: the intruder reaching the destination undetected.

First, we compare how well the intruder behaviors did against the different guard behaviors. Then, we closely examine the effect of the game level layout on the performance of the intruder behaviors. After that, we quantify the effect of modifying the game level layout on our method's performance. We also look into the effect of adjusting the guard's team size

and FOV range on performance. Then, we compare how human players did on this task for one game level and compare it with the intruder methods we defined. Lastly, we report the CPU times our method requires per map.

7.5.1 Success Rate

The main goal of the intruder in our scenario is to reach their destination unnoticed. Figure 7.11 shows a bar chart of the success rate of the different intruder behaviors achieved against the different guard patrol behaviors over all maps we tested. In our previous results, section 3.6 showed that VisMesh had the best coverage out of the three methods; in addition, our human study further confirmed this result in section 4.4.4. The same pattern can be found in this result, where none of the intruder methods could exceed 30% of the success rate against it. Our method achieved around 40% against the other patrol behaviors, which is around 50% better than the intruder who found the shortest path to the destination. However, our method and the shortest path behavior achieved a similar success rate against the VisMesh of 28-29%. In order to better understand the possible causes of this result, we look at the individual results of each map over all patrol behaviors.

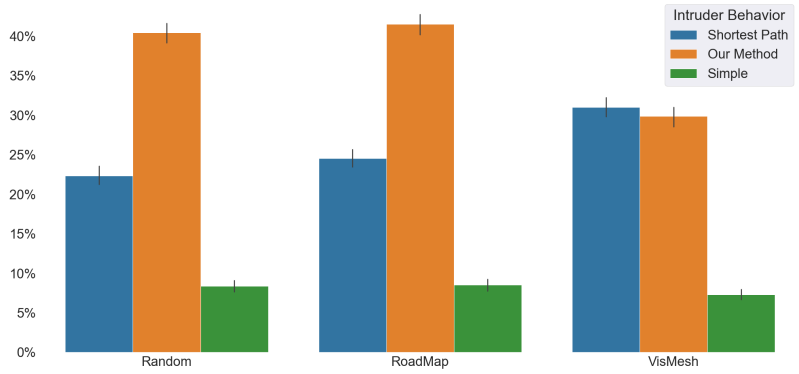


Figure 7.11: The success rate of the different intruder behaviors of reaching the destination without being noticed against the different guard patrol behaviors where the guard team size is four and their FOV range is 10%.

Figure 7.12 shows a bar chart of the success rate for the intruder behaviors against all guard behaviors according to maps. Our method outperformed the baseline methods; however, there was a similar performance between our method and the shortest path method in two levels. The first is "Warehouse", a large map with many intersections. The high count of intersections could have made it more probable for a simple agent to succeed in stealthily reaching the destination. As for the second game level, "Metal Gear Solid" is a smaller level, and in this case, the fixed projection distance of the future trajectories of the guards could have made the intruder in our method more hesitant and stayed stationary longer, and this caused them to be more likely to be detected. As for "Alien Isolation" and "Arkham Asylum", both maps had limited unique paths the intruder could take, making it harder for the intruder to pass unnoticed. We can explore the different projected trajectory properties in future work.

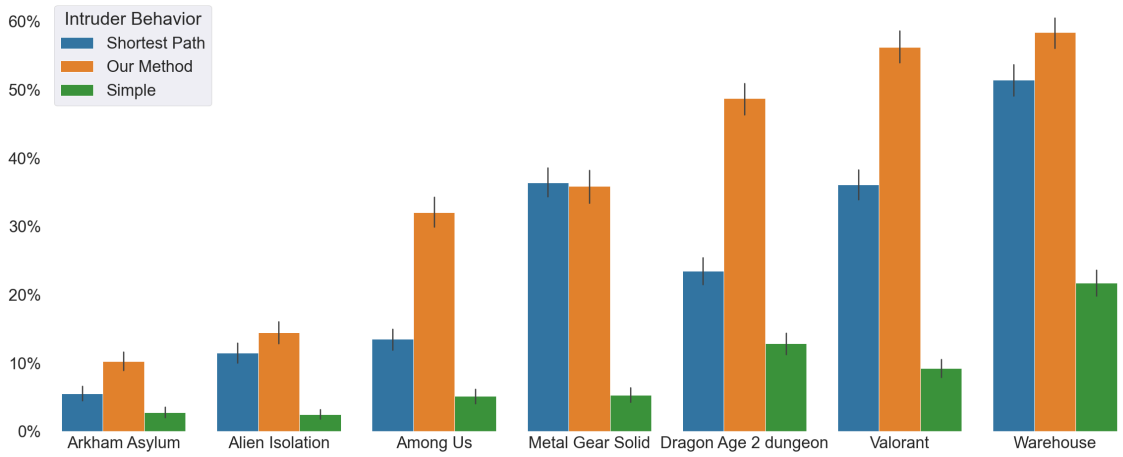


Figure 7.12: The success rate of the different intruder behaviors of reaching the destination without being noticed against the different guard patrol behaviors where the guard team size is four and their FOV range is 10%.

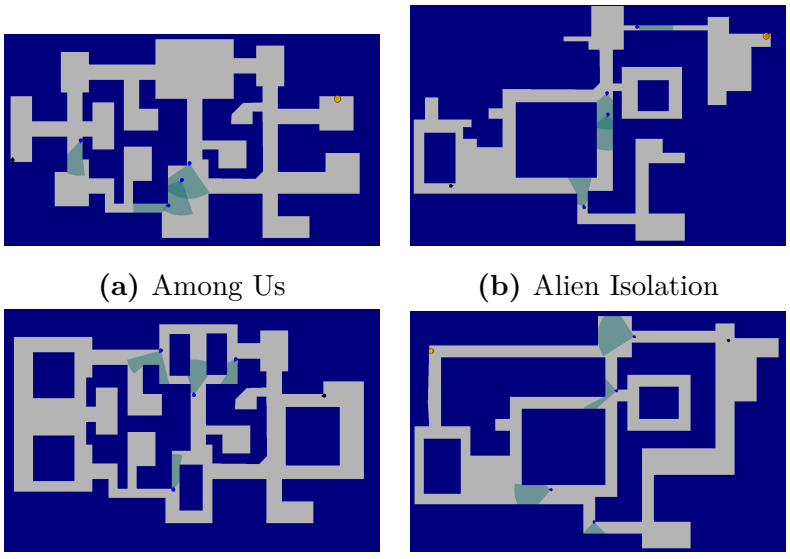
The intruder's task becomes easier for larger game levels, given that we fix the number of guards. For example, in "Valorant" and "Warehouse" maps, four guards were much easier to avoid compared to the smaller levels. Increasing the guard team size might reduce the success rate. Additionally, we observe the difference between the performance of the shortest path

behavior and our method to be more distinct in the "Valorant" map than the "Warehouse". This could be explained by the larger intersection count in the latter map, which gave the guards more routes to take for patrol. This motivates us to understand how changing the game level would affect the intruder's performance.

7.5.2 Map Modification

Our focus lies in examining the map features that influence the effectiveness of stealthy pathfinding. We investigate two primary modification methods. The initial one involves reducing dead-ends and adding cycles on the map, directly impacting the intruder's path options toward their objective. To test this, we choose the "Alien Isolation" map for its low cycle count. The second feature under consideration involves the presence of open spaces within the map. When these open spaces are filled, the map's topology aligns more closely with the corresponding roadmap. This adjustment has the potential to enhance the overall performance of our method. We chose the "Among Us" map as the second map for our testing. Figure 7.13 shows how we modified the two maps.

We ran the same number of rounds against the four guards with 10% FOV. Figure 7.14 shows a bar chart of the success rate between the four maps. The modifications made clear improvements to the success ratio of our method, evidently having more impact than the shortest path behavior. This result shows that such modifications can alter the difficulty of stealth navigation in a specific scenario. Compared to the baseline methods, the diversity of success ratio shows that our method takes advantage of such modifications for more successful stealth navigation. Upon evaluating the performance, we noticed that our method demonstrated a certain degree of "hesitation" due to its safety-first approach. Consequently, we believe conducting further tests with different setups is imperative to enhance the method's performance and fully leverage the benefits derived from map modifications. Further, we explore the effect of simply adjusting the guard's team size or the range of their FOV.



(a) Among Us **(b)** Alien Isolation

(c) Modified Among Us **(d)** Modified Alien Isolation

Figure 7.13: The maps are modified by connecting several dead-ends to create more cycles on the map. Furthermore, we enhanced the occlusion in the "AmongUs" map by adding obstacles in the open spaces.

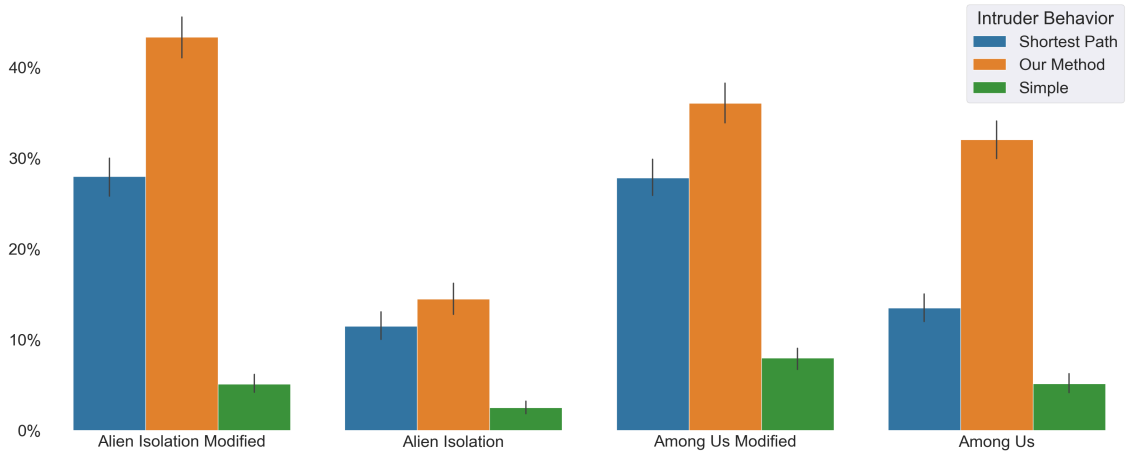


Figure 7.14: Success rate against four guards with 10% for four maps; Alien Isolation, Among Us, and their respective modified versions.

7.5.3 Team Size

The number of guards available significantly impacts the success of stealthy navigation. Given a patrol behavior that efficiently separates the guard routes, more guards usually

result in a difficult task for the intruder. To study the effect of adjusting the team size, we chose three maps of different sizes and layouts. Figure 7.15 shows a bar chart of the success rate of an intruder against guards of different team sizes. Across all maps, we found that the success rate of our method drops at a varied rate. The drop varies depending on the map layout and the previous and current team size. A significant drop rate could be an indicator of a certain difficulty level. For example, in the map "Among Us", the drop rate between four guards and five is larger than the one between five and six guards. We could use this to categorize different difficulty levels based on the drop rate of the success ratio between the different sizes of guard teams.

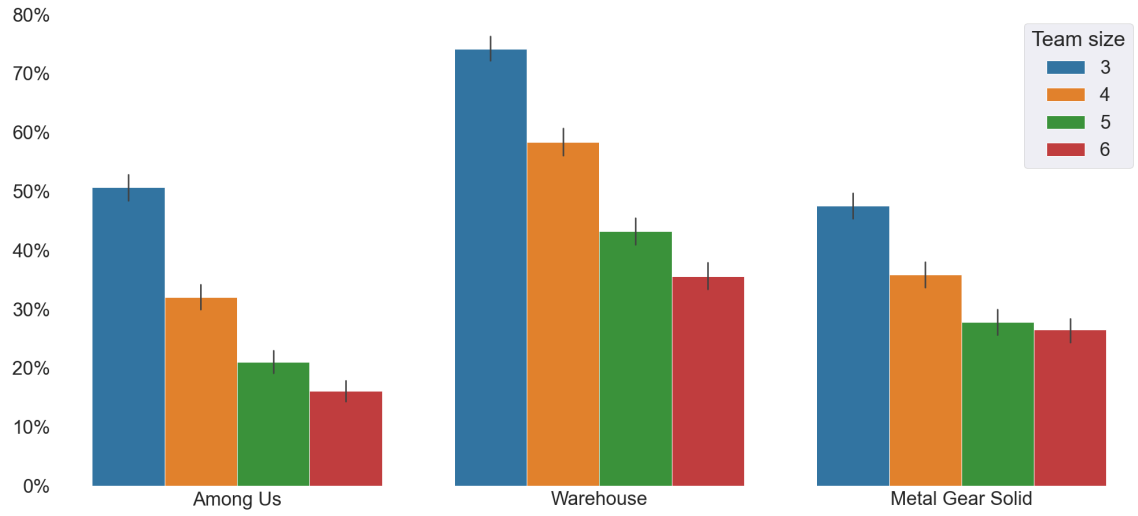


Figure 7.15: Success rate of our method against guards with 10% and different team sizes in three maps.

7.5.4 FOV Range

The FOV range determines how far the guard can "see". Hence, it is also likely to impact the intruder's success in reaching their goal undetected. Figure 7.16 shows a bar chart of the success ratio of our method against guards with a team of four. Increasing the range of the FOV had a more significant impact on the success ratio than increasing the guard count

for a shorter FOV. There are several explanations for this result; one, longer FOV meant a longer projection distance on the roadmap the intruder uses, and this removed more line segments from the roadmap the intruder uses to find a safe path, so it remained more hesitant and stationery. The second reason is that guards with longer FOV cover more than guards with shorter FOV, and shorter FOV are easier to avoid because they need time to reach a position in the distance while a longer FOV can do it instantly.

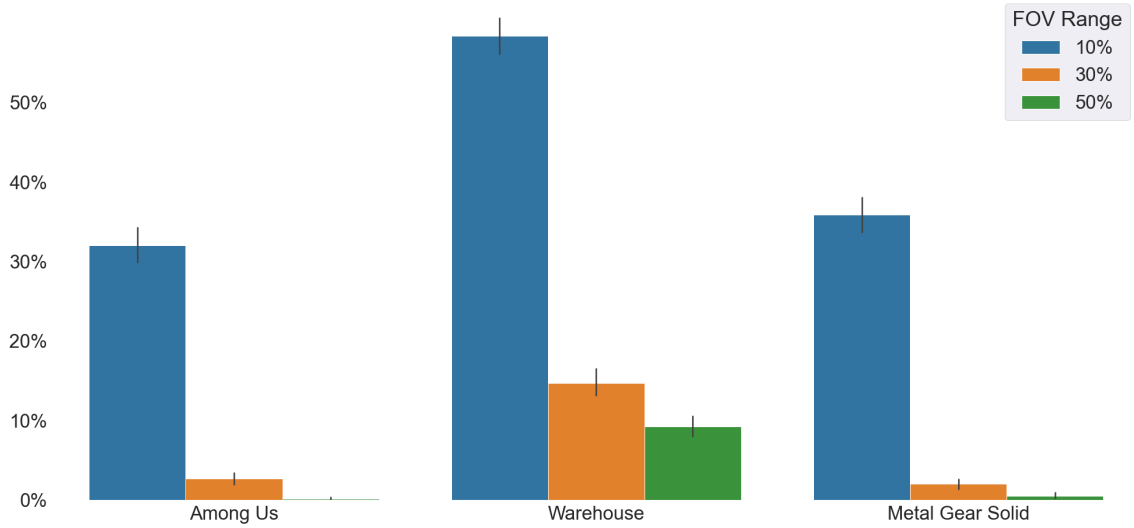


Figure 7.16: Success rate of our method against guards with different FOV ranges and a team size of four in three maps.

7.5.5 Human Comparison

As another benchmark, we compare the human performance in this task from the user study we held in chapter 4. Figure 7.17 shows a bar chart of the success rates against a team of four guards and each team with different patrol behavior in the "Among Us" map. Human players had the highest success against the Roadmap patrol method with a 70% rate, while our method achieved around 30%. The Vismesh was the hardest for all intruders, and against the Random, our method performed the closest to human players' success. Over all the patrol methods, our method had 35% less success than human players. Results show that

human players still significantly outperform our method. Exploring how we can enhance our method's performance by learning from the strategies employed by human players represents an intriguing avenue for future research in this study.

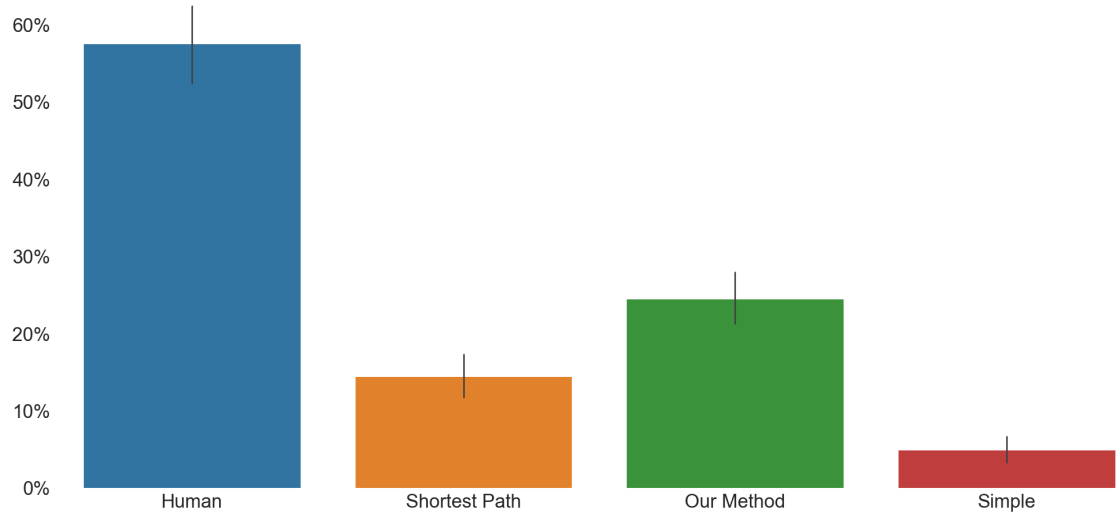


Figure 7.17: The success rate of the different intruder behaviors along with human players of reaching the destination without being noticed against the different guard patrol behaviors where the guard team size is four, and had a FOV of range 10% for the "Among Us" game level.

7.5.6 Computation Costs

While the preliminary skeletal graph is precalculated, our approach demands ongoing changes to the graph structure along with repeated pathfinding operations. Consequently, we measured performance by quantifying the duration it took to implement our algorithm within a single iteration of the game loop. The tabulated outcomes presented in Table 7.1 are mean values derived from numerous iterations conducted during 120-second trial runs, all within the context of a scenario involving four guards.

Name	Time (ms)	Area (m^2)	Edge Count	Node Count
Dragon Age dr_slavers	5	361.8	43	41
Metal Gear Solid: Docks	6	236.3	45	41
Alien Isolation	10	347.6	71	69
Modified Alien Isolation	10	634.2	64	60
Warehouse	12.5	504.3	79	63
Valorant: Ascent	13	552.7	81	71
Among Us	13	437.1	88	87
Modified Among Us	18	432.4	97	90

Table 7.1: Average decision time for different game maps, along with the skeletal graph edge and node count. The experiments were done on a CPU Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz with 16 GB RAM and NVIDIA GeForce GTX 1080 Ti. Dragon Age 2 dr_slavers map was imported from MovingAI. The original map is stored as a grid of 260x315.

The findings demonstrate that our initial implementation is reasonably feasible and adept at providing consistent performance. This is particularly noticeable considering that game AI behavior is commonly executed at a frame rate significantly lower than graphics. Notably, the incurred cost depends on the amount of edges present in the skeletal graph.

7.6 Summary

Implementing a practical approach to stealthy pathfinding can yield improvements in level design, testing, and player engagement. For this, it is important to recognize the impact of various elements on the complexity of covert pathfinding in stealth scenarios. Better stealth pathfinding makes it possible to automate tests for stealth scenarios, rather than relying solely on human playtesting. Additionally, allied NPCs, who require stealthy navigation alongside a player, will be better at preserving player immersion while avoiding potential frustration arising from inferior stealth planning.

Our developed method introduces a versatile real-time methodology capable of managing scenarios characterized by non-deterministic conditions, where observers' movements are not restricted to predetermined patrol paths. This encompasses instances of stealth gameplay wherein enemies can modify their trajectories dynamically, for instance, responding to

alarms. It also pertains to the extensive domain of procedurally generated game environments that produce varied gameplay encounters. Through a thorough analysis of various factors, we shed light on the influence exerted by tuning different parameters. We observe that the inclusion of additional guards consistently leads to a decrease in success rates, albeit within reasonable thresholds.

Chapter 8

Related Work

In this chapter, we discuss and present the related work touching the different subjects presented in this thesis under the umbrella of stealth game AI and player experience. Firstly, in Section 8.1, we investigate topics concerning agent exploration and patrol behavior. Then, in Section 8.2, we explore previous research on how agents can track an opponent's movements in virtual environments. Additionally, in Section 8.3, we delve into the literature on measuring player experience and the impact that NPCs can have on the experience. Finally, in Section 8.4, we review prior work on creating stealthy paths for agents in both robotics and video games.

8.1 Guard Patrol Behavior

The issue of designing patrol routes is primarily concerned with achieving complete coverage of an area. While this problem has been studied in the context of robotics and algorithmic complexity, it has not received much attention in the realm of games. The most well-known theoretical work on area coverage is the Art Gallery Problem (AGP) [89], where Chvátal demonstrated that in a simple polygon with n vertices, $\lfloor n/3 \rfloor$ cameras with 360° infinite field of view are sufficient and sometimes necessary to cover the entire area [31]. Mobile guards are an extension of this problem, known as the Watchmen Route Problem (WRP). The objective

is to determine the shortest path within a polygon that guarantees complete coverage from any point on that path. While solving this problem optimally is NP-hard [29], a solution with a time complexity of $O(n^5)$ exists for the general case where the polygon can be entered from any point [112]. Optimizing guard routes to cover an area can be conceptualized as a set cover problem. Yet, this approach necessitates the division of the game space into discrete units, a process that depends on manual intervention from developers [14].

The literature also presents various multi-agent scenarios related to the coverage problem, such as those discussed in Laguna-Bercero et al. [69] and Ashok et al. [10]. In these studies, guards are assumed to have an unlimited range of vision and a 360° view angle, which may not be practical for most games where agents have limited FOV. A more relevant set of problems for games is the class of "lawnmower problems", which assumes a more limited FOV [8]. These problems approach coverage as a task similar to finding the shortest path for a lawnmower to mow tall grass in a field. Guards with a restricted field of view can be viewed as similar to lawnmowers in this context. However, the efficient solutions to these problems tend to result in dense, zigzag patterns [42]. While these patterns are suitable for achieving complete coverage, they do not reflect realistic human observation behavior, even with a limited FOV.

In robotics research, a widely used method for coverage is a grid-based world representation, which involves dividing the space into a grid of evenly spaced nodes [80]. This approach enables coverage by visiting all accessible nodes, either deterministically or probabilistically [40]. However, the effectiveness of grid-based approaches depends on the grid's resolution, and their completeness is linked to the granularity of the grid. Although it is one of the most straightforward techniques for problems related to space partitioning, it becomes computationally expensive for larger spaces [115].

Using grid-based representation in robotics has facilitated the development of a standard algorithm for exploring an environment [81]. An example of such an algorithm is the *occupancy map* technique, which employs the grid representation of the environment by assigning each node a likelihood of being occupied or possessing other characteristics, such

as explored or traversable [80]. Incorporating occupancy maps in AI has extended beyond robotics research and has been utilized in commercial gaming. An example is the game *Third Eye Crime*, which employs occupancy maps to enhance knowledge representation and create more realistic pursuit and search behavior in stealth scenarios [63]. The game utilizes a probability distribution that diffuses to nearby locations after the intruder is discovered and line of sight is lost, representing the likelihood of the intruder's presence in those locations [62]. This approach results in intelligent guard behavior, but it is only implemented after the intruder is detected. Before that, guards followed pre-scripted patterns, and their patrol logic did not possess intelligent coverage of the game level.

Occupancy maps have also been applied to create dynamic, exploratory behavior for NPCs in video games. For instance, in the turn-based roguelike game NetHack, a grid-based algorithm utilizing occupancy maps has been proposed to direct exploration towards unseen areas, employing a straightforward greedy approach, leveraging the game's pre-existing grid-based representation [24]. Another grid-based approach to exploration was used in an open real-time strategy (RTS) game that took the fog of war into account, where the NPC was guided through *potential fields* [51]. While occupancy maps and related systems offer a simple and robust architecture for coverage and patrol tasks, a significant drawback is their reliance on simplistic discretizations of the game level. In larger and obstacle-rich game worlds, finer-grained grids are required to provide movement flexibility and better conform to the level geometry, which can be computationally expensive. This calls for more effective data structures capable of balancing extensive coverage and computational costs, meeting the real-time demands of modern games.

A limited number of other research works have focused on generating guard patrols in games. One study aimed to create guard movement and patrol patterns by using a generated roadmap at the game level and incorporating a grammar-based route and behavior construction [127]. The approach generated visually interesting results, but the guard paths were relatively simple and short, with no effort to ensure comprehensive coverage. Another approach by the same research group tackled the problem of exhaustive exploration, a related

issue to patrol, by constructing a tour of camera locations generated through the solution of the AGP [30]. Nonetheless, this approach is unsuited for agents with a limited FOV. Another recent study by Seiref heuristically tackled the WRP on grid-based environments [99]. However, this study did not address real-time scenarios, typically necessary for most commercial games.

8.2 Guard Search Behavior

The problem of locating an opponent in a game is contingent on the capacity to track and forecast their position once they have moved out of sight. This issue has been tackled in various domains, including military surveillance for tracking enemy positions [18], monitoring human movement trajectories in robotics [12], using reinforcement learning for robot search planning [72], and enabling NPC to display human-like motion in First-Person Shooters [57]. Additionally, prior research has utilized a Turing "hide-and-seek" test to assess the performance of agents playing such a game in a room [25].

Opponent tracking is a central feature of the gameplay in *Third Eye Crime*, a well-known game that used a modified version of *occupancy maps* as its core mechanic [62, 63]. The game map is overlaid with a grid, and each grid node has a numerical variable that represents the probability of an opponent occupying that node. In a typical game scenario, when the opponent is detected by a guard's FOV, all guards take the shortest path to the opponent's position. However, once the opponent is out of sight, the node corresponding to the position they were last seen has a probability of 1. The update function uniformly diffuses the probability across neighboring nodes and normalizes them as time passes. Guard roles are to cover the node with the highest probability, and once a guard sees a node, its probability is reset to zero. While this method results in interesting guard behavior, the accuracy of tracking an opponent's position depends on the granularity of the grid used, with lower resolution grids leading to less accurate tracking and higher resolution grids increasing computation and memory costs [35].

The shortcomings of occupancy maps were addressed by using particle filters, a technique originally developed in robotics [116]. By using this method, there is no need to discretize the space into a grid, like in occupancy maps [13]. In general, this method works by sampling a finite set of weighted particles that represent the possible positions of the opponent. The weight of each particle corresponds to the probability the opponent is in that position. Once a particle is seen, it is removed, and unseen particles are resampled. After that, the new particles are updated by a random walk by choosing a direction; then, it is moved in that direction at max speed for a one-time step. The memory requirement for using particle filters is, unlike occupancy maps, independent of the size of the map. Alternatively, the computational expense relies on the number of particles sampled; a high number of particles results in high accuracy in exchange for performance, and a low particle count will result in lower accuracy.

Particle filters and occupancy maps have different computational and memory requirements, with particle filters typically requiring less. However, both methods have a common weakness: they assign equal weights to unlikely opponent positions. This trait is due to the randomness of particle filters and the omnidirectional probability diffusion in occupancy maps. Essentially, neither method takes advantage of the game level's layout to improve the accuracy of position weighting. To address this limitation, a motion model known as "simulacra" was developed [35]. Simulacra defines particles based on a navigation graph used for NPC pathfinding and uses restricted sampling to create particles that more accurately reflect NPC movements on the graph.

While the simulacrum approach has been algorithmically described, there has been limited research to evaluate its characteristics or determine the effects of relying solely on a navigation graph for both NPC movement and probability propagation. Our design employs a similar approach but with some enhancements. Specifically, we use a separate graph that captures the map properties independent of the NPCs' pathfinding system. Additionally, we develop a multi-agent framework for this problem.

8.3 Player Perception

Studies have shown that players tend to enjoy playing against NPCs that exhibit more human-like behavior [103]. Consequently, there have been multiple efforts to assess player perception of AI behavior. For example, competitions have been organized to test human-like opponents [117], which can be seen as a type of "Turing test" for game NPCs [121]. The 2K BotPrize Contest was one such competition, which evaluated bots for the First-Person shooter game Unreal Tournament 2004. In this contest, human players were asked to judge whether they were playing against a human or a bot. Although none of the bots passed the test, some of them managed to confuse humans in their judgment by exhibiting a certain degree of error and randomness [56].

In another study, researchers took a different approach to this problem. They asked bots and human players to predict player positions on a simplified top-down presentation of Counter-strike. The study used a particle filter and hidden semi-Markov models, and the results showed that both players and bots made similar numbers of correct and incorrect judgments [57]. This suggests that it may be possible to create more human-like behavior in bots. However, a survey of video game professionals revealed concerns that players might not perceive the increased complexity of NPCs or understand their behavior easily. This phenomenon is known as "the black hole of AI" [66], underscoring the need to evaluate the limits of this phenomenon.

Several studies have explored different aspects of NPC behavior in modern commercial games that could impact player enjoyment. Lankoski & Björk propose various design patterns that enhance the believability of NPCs. These patterns, such as the indication of intentionality, are examined by scrutinizing a single character from the game *Oblivion* [70]. Meanwhile, Warpefelt & Stråat tackle the same matter by analyzing the behavior of NPCs in several role-playing and first-person shooter games [122]. They discovered that immersion is compromised when NPC knowledge is not adequately constrained, and environmental awareness is not properly considered.

Various studies have attempted to focus on player enjoyment directly. For instance, certain studies have investigated the prospect of refining games by creating mathematical models that determine the advancement of a game. The purpose of this is to establish a progress model that can adjust player satisfaction by "accelerating" game progress in sports and board games [60, 109, 110]. In addition, player enjoyment has been examined in NPC AI for various game genres, such as turn-based strategy games [124], board games [60], and first-person shooters [56, 103].

In contrast, some studies have endeavored to employ learning NPCs to anticipate player satisfaction by defining an "interest" value based on a collection of metrics. In this approach, player enjoyment is evaluated by asking players about their experience after playing each NPC behavior variation [129]. Although there is limited research on player enjoyment in stealth games, they also explored a similar issue in the form of a predator/prey scenario, which shares certain characteristics with our studies [128]. Their research centered on an adapted variation of the identical problem within a narrower context, similar to *Pac-Man*. The objective was to assess the precision of quantitative assessments linked to enjoyable attributes. They attempted to formulate a quantitative gauge for enjoyment; nonetheless, they didn't explore whether particular elements or blends of AI techniques and design held a significantly greater appeal to players.

Even though NPC behavior directly influences player experience, several other elements can also impact that experience. For instance, dialogue lines or "barks" are frequently utilized in commercial games to enhance the overall game experience [94]. These dialogue lines usually consist of a pre-scripted set of lines that are played to players in specific situations. Despite several studies exploring the possibility of generating dialogue using natural language generation [23, 67], the existing literature lacks research that examines the relationship between the content conveyed in the dialogue, NPC behavior and how these two factors collectively affect the game experience. In our study, we delve into the influence of various types of "barks" on player enjoyment in response to diverse enemy behaviors.

8.4 Stealthy Path Finding

The issue of covert pathfinding has been previously addressed in the context of robotics, with numerous initial approaches relying on discretization. Marzouqi and Jarvis, for instance, discretize the walkable space into a grid, where each cell is assigned a numerical value that denotes its visibility from the standpoint of other cells [74]. This representation allows a robot to devise a covert path by traversing the least visible cells, although this approach fails to consider the existence of dynamic observers. Dealing with the latter challenge involves solving the computationally complex problem of optimizing a time and space-constrained motion problem [114], which can be resolved through adequate approximation and parallelization but is impractical for real-time game contexts.

Many variations of the problem exist, depending on the assumptions of how much of the state must be observed or inferred vs. assumed. Sukhatme and Mataric, for instance, incorporated the existence of a stationary observer, while stealthy robots shared the same environment representation [108]. Johansson and Dell'Acqua assume a more realistic context in which the observer's position is unknown, and an agent must use their observations to build a probability distribution of the observer's position [65]. Their method was computationally expensive, with each update requiring nearly a second to lay out a stealthy path. This means this method is inapplicable for real-time games with a limited time budget of milliseconds. One method focused on real-time strategy (RTS) game applications involves defining safe paths using influence maps to avoid and ambush adversaries [34]. Hladky and Bulitko introduced a method that used a particle filter approach to predict opponent positions in partially observable environments [57]. This method can potentially augment an additional feature to avoid these opponents; however, this method requires historical data.

Discretization based on a grid has the drawback of being relatively memory intensive. A more efficient method is to use natural space decomposition. For instance, Mendonça attempted to find a covert path using a custom NavMesh and assigning a weight to each polygon based on its proximity to cover [77]. Brewer also utilizes NavMeshes to allocate tactical waypoints, including cover points [21]. To reduce the use of many specialized pathfinding

methods, our work avoids modifying the NavMesh for the purpose of the covert agent. Other studies have exploited the level's geometry by staying in the safe, occluded region of the observer's current FOV, assuming known future observer positions, thereby creating interesting stealthy behavior [101].

Another approach to creating stealth path planning against multiple stationary observers involves using *corridor maps* for navigation, as suggested by previous research [90]. This method alters the cost value of an edge in a graph depending on the extent to which it is occluded from the observers' field of view [44]. Our work builds upon this graph-based approach, adapting it to accommodate mobile observers.

Suppose the patrol pattern of a moving observer is static and known. In that case, the easiest way to handle them is through using A* or a more efficient heuristic search technique like Rapidly-exploring Random Tree (RRT) to search the future, time-extruded state-space and find a path that avoids both obstacles and observer observation [119]. Further work has extended this approach to incorporate combat and level analysis [120], body-hiding [38], and the use of distractions [17]. Other variations of the problem, such as interception, have also been studied in the field of robotics [91]. However, the deterministic nature of the observer behavior limits the application of these methods to less structured contexts, such as games that rely on randomization for replayability or those that use procedural generation.

Chapter 9

Conclusions & Future Work

Finding possible methods to capture relevant features about the game level can enhance the effectiveness of NPC AI. This dissertation suggested several methods to present space to create novel NPC behaviors for stealth game scenarios. Additionally, we evaluated the user experience when playing against these methods. This chapter describes the main findings from our empirical experiments and user studies and the possible future work in this field.

9.1 Conclusions

Designing stealth games has its unique set of challenges to overcome, and introducing computational methods that can reduce the time and effort required can be beneficial. In this thesis, we introduced and tested several applied techniques that reduced the need for design efforts in stealth games.

Guard patrol patterns form mobile obstacles players aim to avoid. Game designers usually craft these patterns by placing static waypoints on the game level so guards cyclically follow them. We introduced several methods that create dynamic guard patrol behavior in games without hard-coded waypoints. We believe that this will offer numerous benefits that enhance gameplay and immersion.

By allowing guards to adapt their patrol routes based on the game level, the AI becomes more realistic and less predictable. This leads to a more immersive and challenging experience for players facing adaptive and responsive enemies. With guards taking different paths and responding differently, players cannot rely on memorizing guard patterns. This forces them to think on their feet and adapt their strategies, introducing complexity and variety to the gameplay. Moreover, the lack of fixed waypoints increases the replayability of the game. Each playthrough becomes a unique experience as guard behavior changes, offering new challenges and encounters. This replayability keeps players engaged and motivated to explore different approaches in subsequent sessions. By avoiding hard-coded waypoints, game developers can future-proof their design. Instead of manually scripting every guard patrol route, they can create rules and parameters that guide the AI's decision-making. This allows for easier content updates and level modifications without extensive reprogramming.

As an extension of our work, we explored dynamic search behavior for guards searching for a player they discovered earlier. This is a crucial mechanic in several stealth games. We expanded this work in our user study by investigating the impact of dialog lines on player experience. Our results hold significant implications for game design.

Game developers can consider incorporating contextual dialogs to enhance player engagement and enjoyment, even in scenarios with simpler enemy behaviors. Moreover, the study highlighted the role of contextual dialogs in enhancing player enjoyment of the game. The presence of contextual dialogues might also inspire further advancements in the field of AI for games. Game developers could explore ways to implement such dialogues and behaviors, crafting more dynamic and believable AI opponents. Another notable outcome of the study is the potential efficiency in game development. If players derive similar enjoyment from simpler guard behaviors with contextual dialogs, developers could focus more on crafting engaging narrative elements and allocate fewer resources to complex AI behaviors without sacrificing player satisfaction.

While the role of stealthy path planning is typically taken by players, creating an NPC capable of playing stealth games can have far-reaching benefits and applications. For exam-

ple, game developers can use such NPCs to evaluate the balance, challenge, and level design of their games. They can provide valuable feedback on potential exploits, game-breaking strategies, or overly difficult sections, leading to more refined and enjoyable gaming experiences. Additionally, stealthy friendly NPCs can benefit gameplay since making them completely invisible to enemies can lead to immersion-breaking situations. Our work can assist in improving their stealth path planning so they might use cover effectively without rendering them completely invisible to enemies. Having more reliable stealthy allies can improve player immersion and enjoyment.

9.1.1 Research Questions

In response to the research questions we posed in Section 1.1, we summarize our findings as follows:

Dynamic Guard Patrol

1. **What are the possible ways to produce real-time dynamic patrol behavior?**

   We presented three primary dynamic patrol behaviors, each associated with a distinct game level representation. These approaches are a grid-based model, a straight skeleton-based model (RoadMap), and a space-decomposition-based (VisMesh) model.

2. **How different are they in terms of efficiency and computational cost?**

   We discovered that each approach is capable of generating dynamic patrol behavior, each accompanied by its distinct features and advantages. However, the VisMesh method outperformed others, particularly in its consistent coverage of the game level. In terms of computational efficiency, we noted that the grid-based representation proves impractical for real-time applications, especially in larger map scenarios. Contrarily, the remaining two methods present notable memory efficiency advantages, rendering them better-suited options for larger maps.

3. **Do these behaviors contribute to the overall enjoyment of the game?**

Players exhibited differing degrees of enjoyment when engaging with various patrol behaviors, favoring RoadMap and VisMesh over Random patrol. Furthermore, players perceived VisMesh as the most effective and difficult patrol behavior, displaying characteristics similar to search-like behavior.

4. **What general traits in patrol behavior affect player enjoyment?**

   We observed that many players found enjoyment in engaging with the VisMesh behavior, even if their in-game performance was least compared to the other behaviors. A deeper examination of their feedback revealed that the most challenging behaviors were the most motivating for them to enhance their gaming skills. Additionally, players highlighted that the unpredictability of these behaviors heightened their overall game enjoyment, with VisMesh being singled out as the most unpredictable. These trends suggest the need for further investigation through additional testing.

Guard Search and Dialog

1. **Is it possible to create a feasible, credible, and interesting but still solvable dynamic guard search behavior?**

   We introduced an innovative approach that allows guards to exhibit real-time search behavior for intruders while effectively utilizing their awareness of the level's geometry. This method involved the utilization of a skeletal graph representation to propagate the probability of potential opponent locations. In our experimentation across various maps from existing commercial games, we observed intriguing behavior patterns in multi-guard searches for intruders. While our grid-based variation outperformed this method overall, it demonstrated comparable performance in situations with relatively high levels of occlusion. Additionally, it was proven to be suitable for real-time implementation, resulting in better CPU time efficiency, especially in the case of larger maps.

2. **Can players distinguish complex search tactics, and how does their perception of this relate to presentation components, like simulated spoken interactions from the NPCs?**

   Our exploration through a user study of a stealth game revealed that players can detect variations in search behaviors, particularly when these behaviors stand out as unique. This phenomenon becomes even more pronounced when the behavior seems inconsistent with the expected knowledge that NPCs should logically possess within the game world. Notably, this observable distinction persists even when NPCs employ "barks" that create the illusion of intelligence that fits the context. On the contrary, simpler strategies like randomization can effectively complement well-crafted contextual barks, resulting in an overall positive gaming experience that can rival or even surpass the engagement achieved by deploying more sophisticated NPC behaviors.

Stealthy PathFinding

1. **How can we create stealthy pathfinding accommodating guards with non-deterministic patrol patterns?**

   We developed a method that introduces a versatile real-time stealthy navigation capable of managing scenarios characterized by non-deterministic guard movements and not restricted to predetermined patrol paths. This encompasses instances of stealth gameplay wherein guards can modify their trajectories dynamically, for instance, by responding to alarms. It also pertains to the extensive domain of procedurally generated game environments that produce varied gameplay encounters.

2. **What are the different aspects of the game level or guard team that can affect the success of this method?**

   By conducting a comprehensive examination of several variables, we reveal how adjusting different parameters can impact the performance of our method. Our findings indicate that, compared to modifying the guard's FOV, consistently adding extra

guards results in a finer reduction in success rates. When it comes to the game level, increasing the number of available routes and eliminating dead ends had a more significant positive effect on our method's success rate compared to obscuring open spaces within the game level.

9.2 Future Work

This thesis introduced several improvements to dynmaic NPC behaviors in stealth or action games. However, the domain of this topic is too large for this thesis to cover. In this section, we explore the possible future directions we can take to expand on this work.

9.2.1 Guard Behavior

In our exploration of guard motion in stealth and action games, we have identified several areas to expand our work. One direction involves utilizing reinforcement learning to control the weights that determine the search and patrol behaviors of guards. By incorporating the game level properties, the number of guards, and the FOV radius as inputs, the model can adjust the weights based on these parameters, optimizing guard performance.

Numerous current games depict character motion in a 2D fashion, even though they are visually presented in a 3D environment. Nonetheless, certain games offer players additional choices for movement, enabling vertical actions such as climbing, using grappling hooks, or executing jumps. Adapting our model to accommodate the verticality of 3D game environments poses a challenge. While expanding it to a 3D setting is possible, addressing the issues related to verticality is non-trivial. To create interesting search and patrol behaviors that consider the vertical dimension, we can explore approaches such as creating multiple layers of 2D presentation or modifying the existing structure by adding a depth dimension.

Performance optimization is another area of interest for us. Although our method has demonstrated efficient real-time capabilities on average-sized game maps, we believe there is room for improvement through parameter fine-tuning. This includes optimizing the roadmap

structure, segment size, graph density, and update rate based on the specific map's characteristics. Additionally, for tasks like patrol, calculating patrol routes using one of our methods and having guards follow these routes repeatedly can enhance performance without relying on the defined structure.

Based on user studies conducted on guard patrol, participant feedback highlighted positive aspects of different components belonging to various patrol methods. To address these favorable traits, we propose combining the structure we use to create such behavior. For instance, employing the roadmap in the main parts of the game level and implementing a partial VisMesh system for deadends can provide guards with better separation in the main areas while exhibiting meticulous search behavior in rooms, which players appreciate.

Lastly, we are interested in exploring techniques that minimize intelligence requirements while maintaining a seamless and realistic impression of NPC actions. This may involve modeling player attention in relation to the impact of NPC behavior, which could uncover opportunities to reduce development efforts. Analyzing and examining techniques employed in practical games would contribute to a deeper understanding of the effectiveness and implementation of such approaches.

9.2.2 Intruder Behavior

Developing efficient and viable covert behavior remains an ongoing area of research. There are multiple paths for expanding our work in the future. To improve the adaptability of our method to various conditions, we propose using reinforcement learning techniques and training the intruder on multiple game level layouts and guard setups. Considering several parameters, including map properties, this approach can result in more efficient and engaging behavior.

Exploring different and more efficiently conforming map abstractions would also be valuable. Although we employed the skeletal axis transformation for its simplicity and effectiveness in creating a roadmap, the success of our approach depends on how well the graph covers the actual movement space. Formalizing this property and designing extended abstractions

that ensure good conformance could enhance success rates and enable more efficient propagation without the need for intermediate temporary node construction.

Moreover, evaluating the perception of these methods by human players is crucial. Conducting user studies that involve human players collaborating with stealthy AI companions would provide deeper insights into how the performance of the stealthy AI affects the overall player experience.

Appendix A

Game Level Layouts

In this appendix, we illustrate the maps we used in this thesis. Additionally, we quantify the properties of each map according to the following measures:

- Number of edges: This is the number of edges that belong to the corresponding straight skeleton of the game level.
- Area: This is the area of the walkable space of the game level.
- Radius: This is the minimum among all the maximum distances between nodes of the corresponding straight skeleton of the game level to all other nodes. The radius represents the size or reach of the graph from the perspective of a single node.
- Diameter: This is the longest distance that needs to be traversed to go from one node to another in the corresponding straight skeleton. The diameter can provide a sense of the longest distance that needs to be taken to travel from one point to another on the game level.

Table A.1 shows the properties of the game maps we included.

Table A.1: List of maps and their respective straight skeleton graph properties, accompanied by the grid dimensions for each map.

Name	Edges	Area	Radius	Diameter	Grid Dimensions
Metal Gear Solid	45	236	32	68	60×32
Alien Isolation	67	246	44	101	78×52
Dragon Age 2 dungeon	44	362	50	105	84×70
Among Us	88	437	50	101	84×48
Warehouse	79	504	50	97	72×50
Arkham Asylum	87	620	64	127	113×68
Valorant	78	1020	60	116	104×88

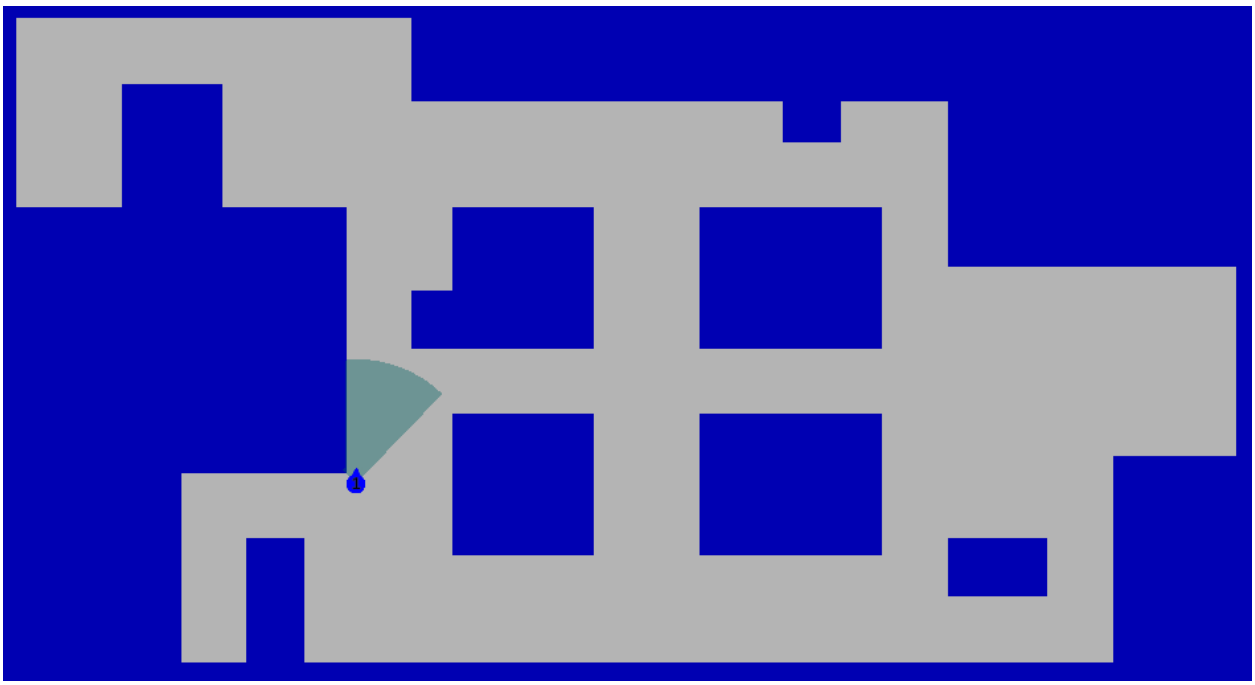


Figure A.1: The Docks map from the commercial game Metal Gear Solid [68]

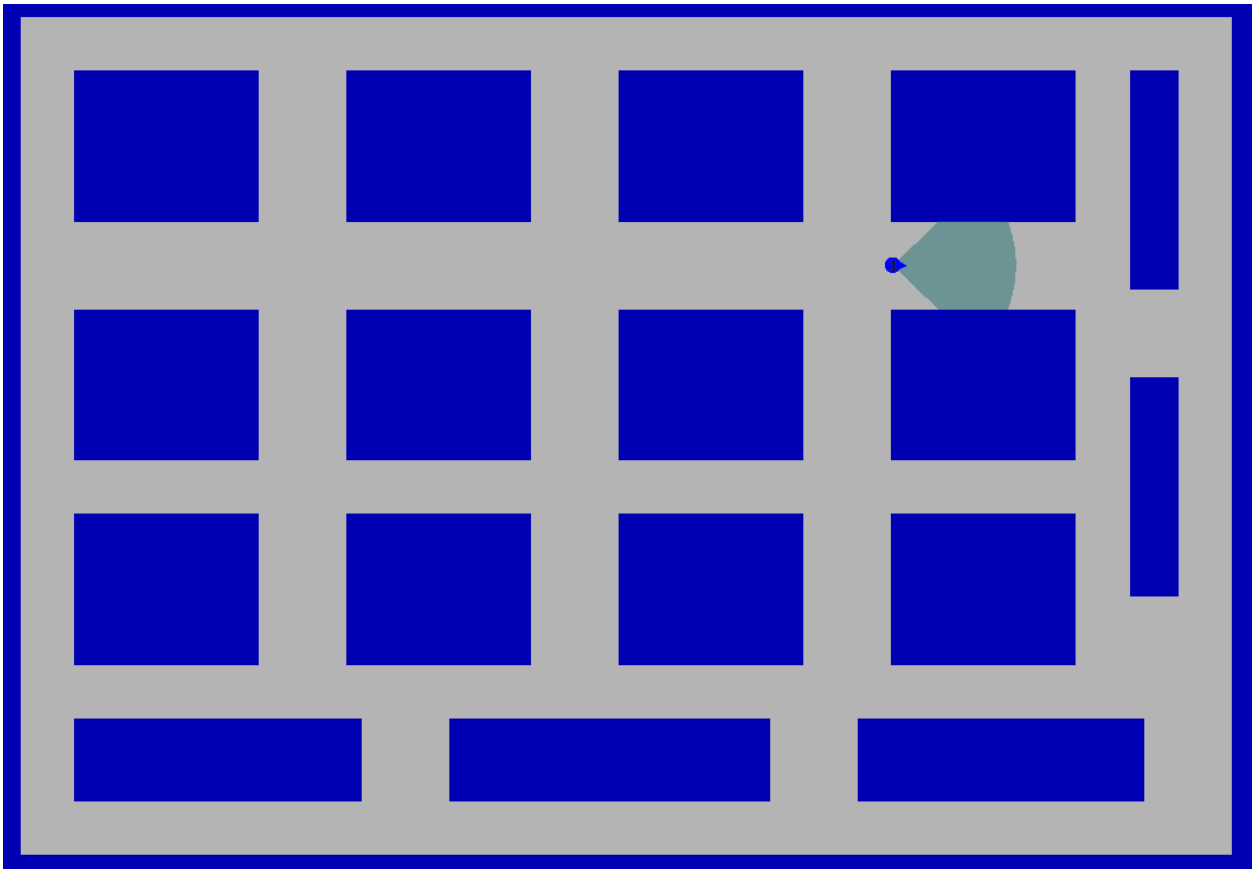


Figure A.2: A map that resembles a Warehouse layout.

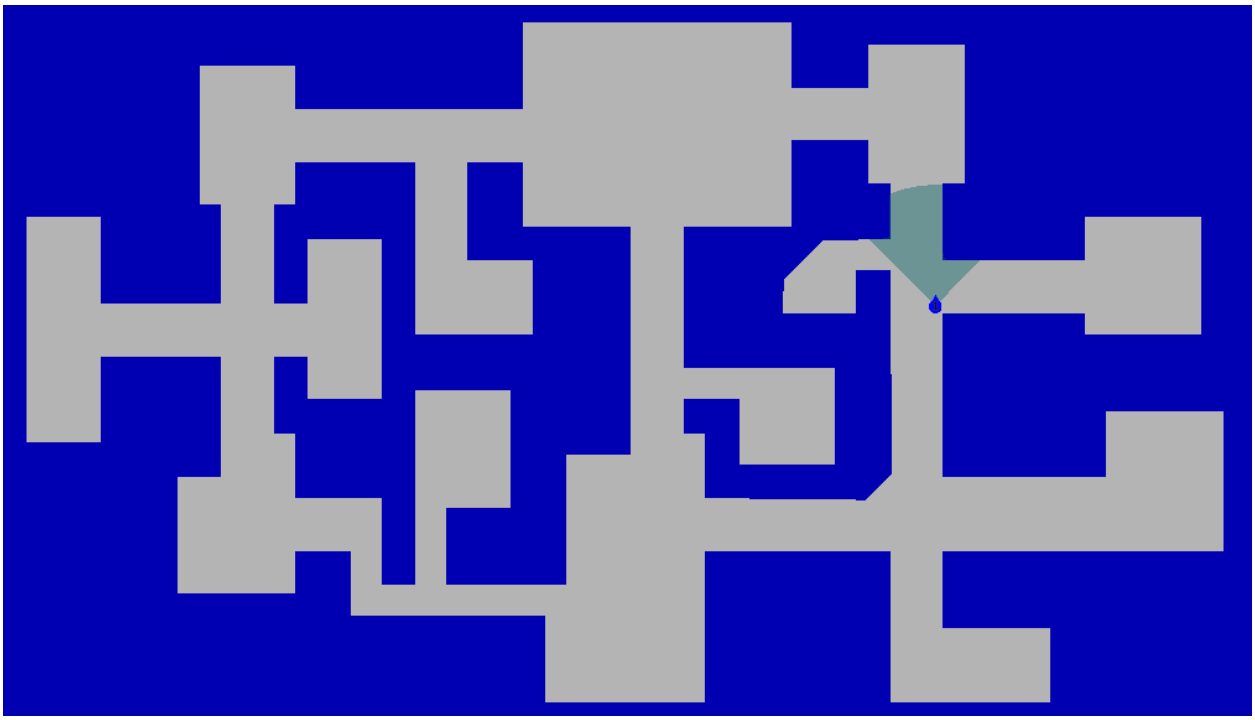


Figure A.3: The Skeld map from the commercial game Among Us [61]

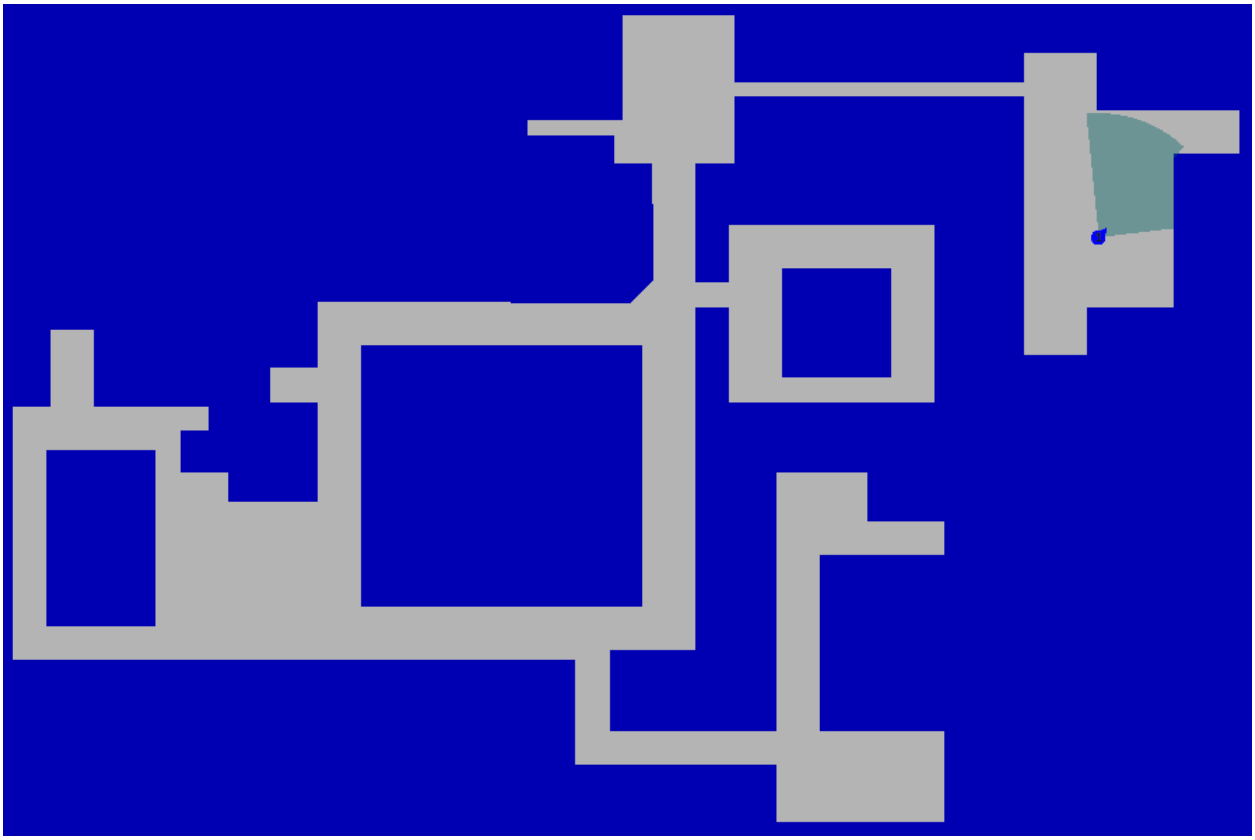


Figure A.4: San Cristobal Medical Facility from Alien Isolation [11]

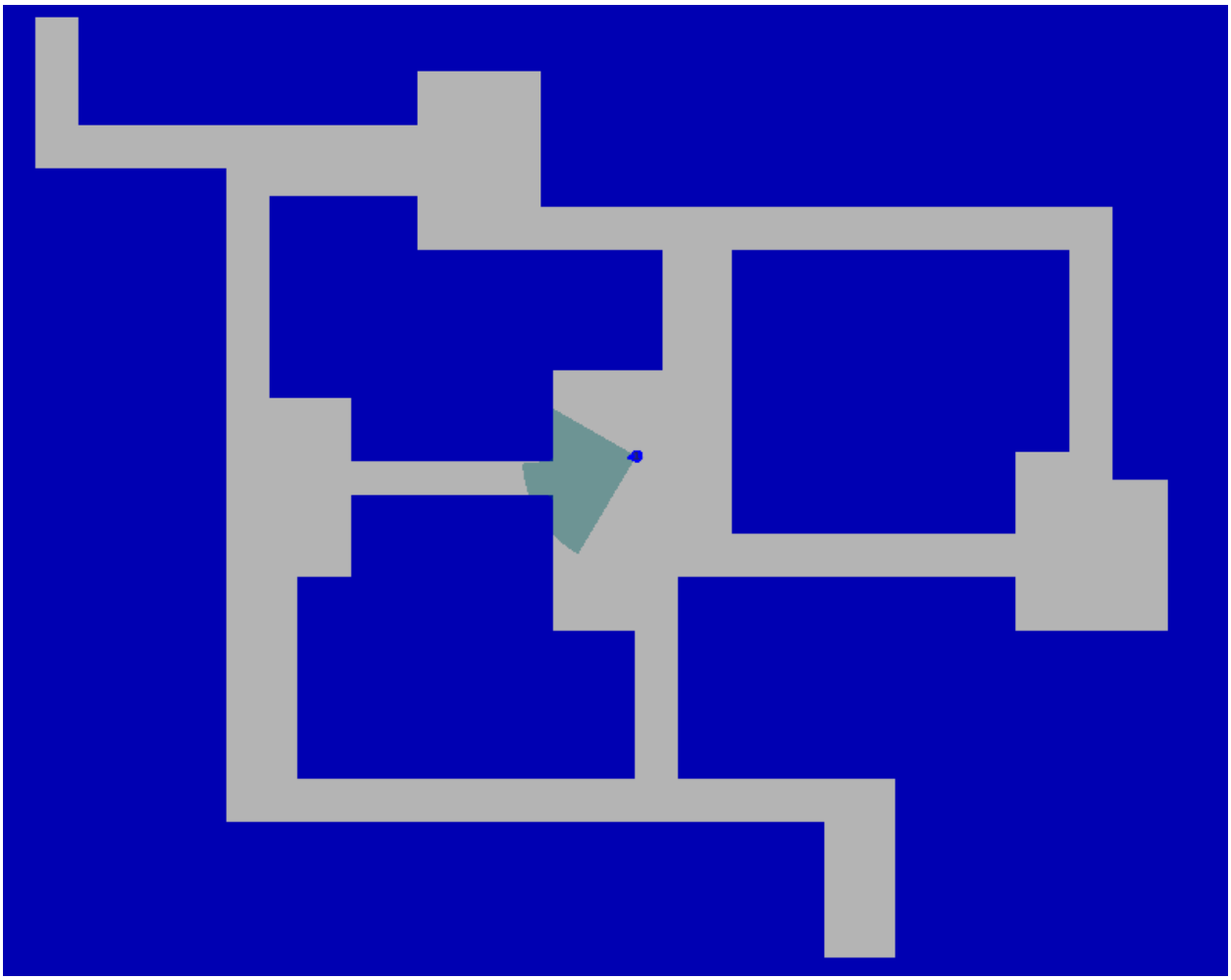


Figure A.5: The “dr_dungeon” map from Dragon Age 2 [15]

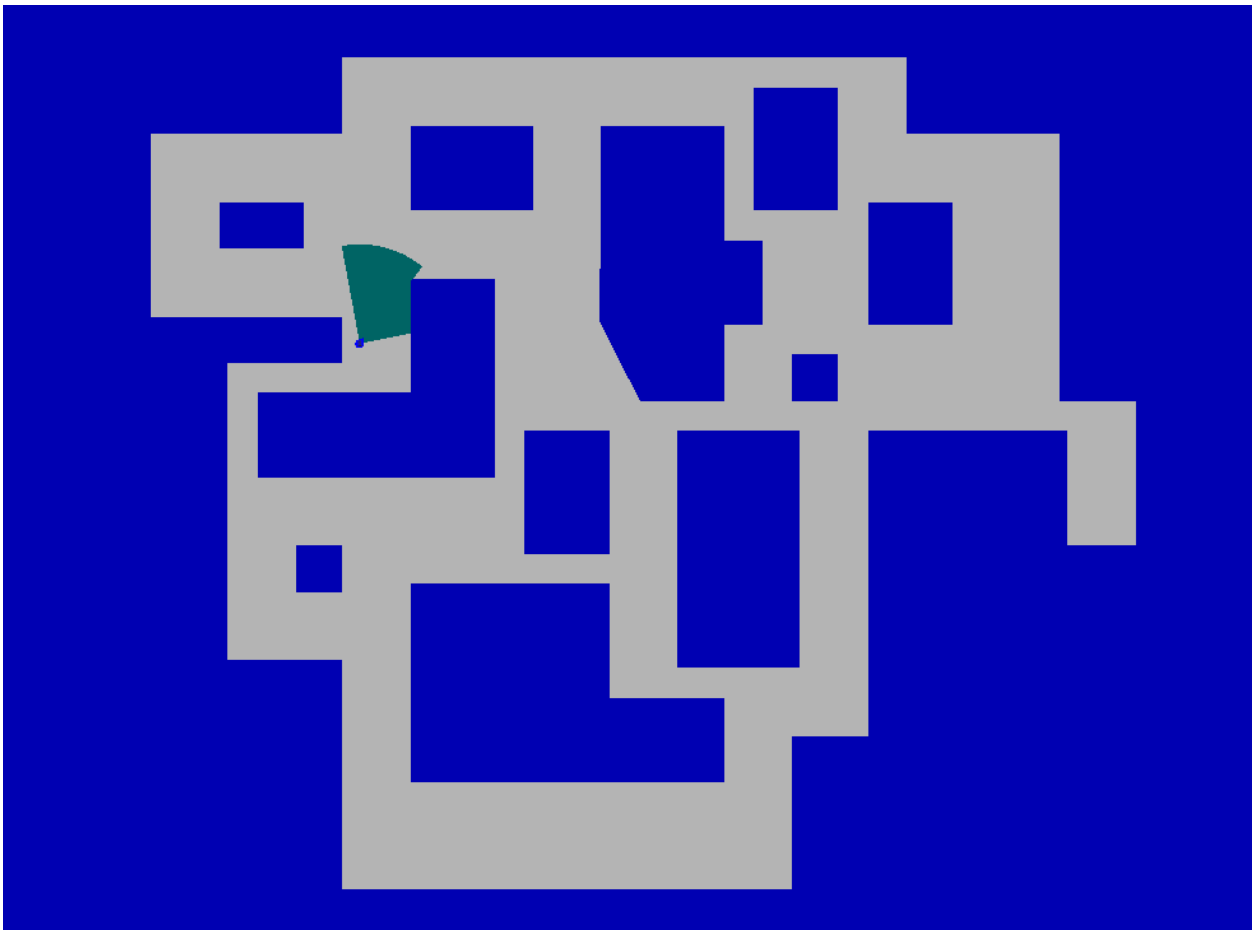


Figure A.6: The “Ascent” map from the game Valorant [95]

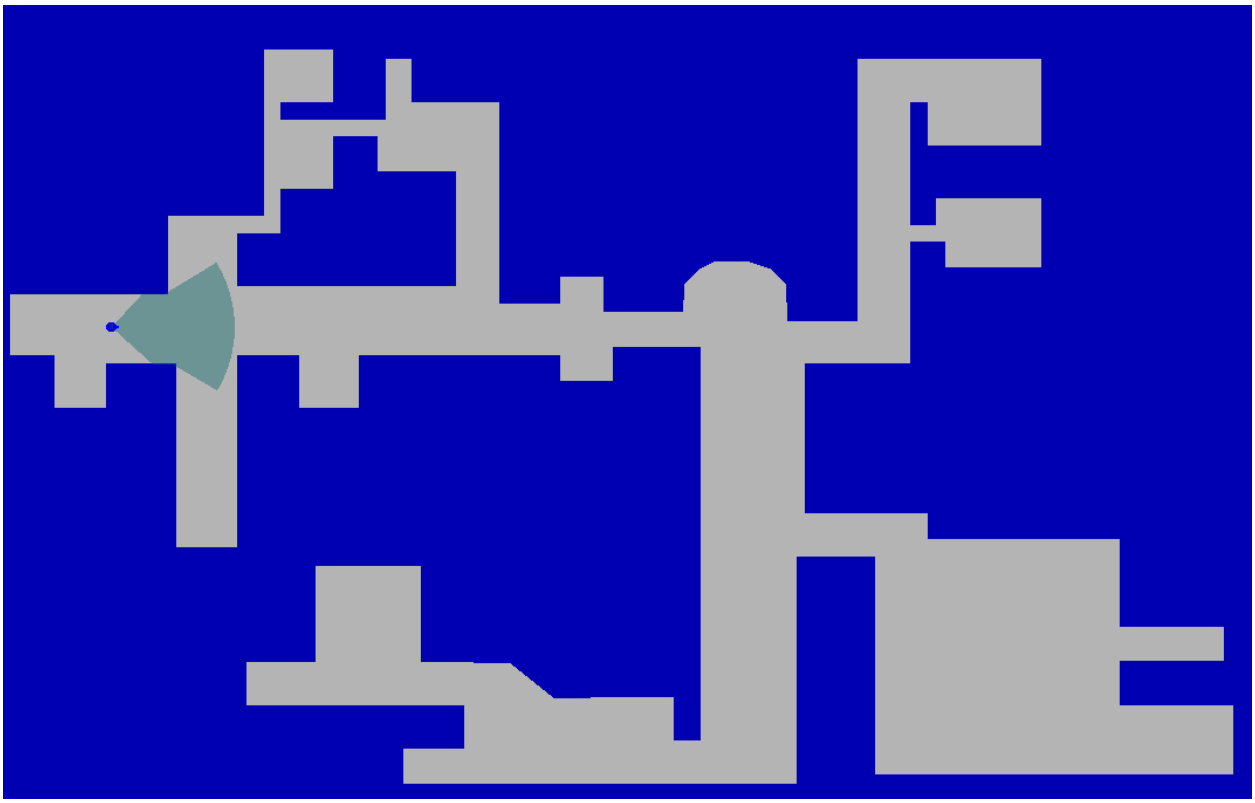


Figure A.7: The Arkham mansion map from the commercial game Batman: Arkham Asylum [105]

Glossary

coverage threshold The threshold to reset a covered region in the VisMesh implementation. Page: 58

covered region A polygon that represents the area guards covered with their FOV over a period of a time. Pages: xiii, 57, 58, 63, 76, 78–80, 83

game level diameter The longest shortest path distance between two points in the game level. This is used as denominator to normalize distances used in heuristics. Pages: 50, 54, 62, 71, 121, 128, 151, 179

straight skeleton A geometric construct derived from a polygon that captures the evolution of the polygon's shape as its edges move inwards at a constant speed, eroding the polygon. Pages: 51, 83, 112, 122, 171, 215

Unity3D A commercial game engine used to create computer games. Page: 46

Bibliography

[1] Activision. Tenchu: Stealth assassin, 1998.

[2] Aichholzer, O., Aurenhammer, F., Alberts, D., and Gärtner, B. *A novel type of skeleton for polygons*. Springer, 1996.

[3] Al Enezi, W., and Verbrugge, C. Dynamic guard patrol in stealth games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (2020), vol. 16, pp. 160–166.

[4] Al Enezi, W., and Verbrugge, C. Skeleton-based multi-agent opponent search. In *2021 IEEE Conference on Games (CoG)* (2021), IEEE, pp. 1–8.

[5] Al Enezi, W., and Verbrugge, C. Stealthy path planning against dynamic observers. In *Proceedings of the 15th ACM SIGGRAPH Conference on Motion, Interaction and Games* (2022), pp. 1–9.

[6] Al Enezi, W., and Verbrugge, C. Evaluating player experience in stealth games: Dynamic guard patrol behavior study. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (2023), vol. 19.

[7] Al Enezi, W., and Verbrugge, C. Investigating the influence of behaviors and dialogs on player enjoyment in stealth games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (2023), vol. 19.

[8] Arkin, E. M., Fekete, S. P., and Mitchell, J. S. Approximation algorithms for lawn mowing and milling. *Computational Geometry 17*, 1-2 (2000), 25–50.

[9] Arkin, R. C. Path planning for a vision-based autonomous robot. In *Mobile Robots I* (1987), vol. 727, International Society for Optics and Photonics, pp. 240–250.

[10] Ashok, P., and Reddy, M. M. Efficient guarding of polygons and terrains. In *International Workshop on Frontiers in Algorithmics* (2019), Springer, pp. 26–37.

[11] Assembly, C. Alien isolation, 2014.

[12] Bennewitz, M., Burgard, W., Cielniak, G., and Thurun, S. Learning motion patterns of people for compliant motion. *International Journal of Robotics Research* (2004).

[13] Bererton, C. State estimation for game ai using particle filters. In *AAAI workshop on challenges in game AI* (2004).

[14] Berger, B., Rompel, J., and Shor, P. W. Efficient nc algorithms for set cover with applications to learning and geometry. *Journal of Computer and System Sciences 49*, 3 (1994), 454–477.

[15] BioWare. Dragon age 2, 2011.

[16] Blum, H. A transformation for extracting new descriptions of shape. *Models for the perception of speech and visual form* (1967), 362–380.

[17] Borodovski, A., and Verbrugge, C. Analyzing stealth games with distractions. In *Twelfth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2016)* (October 2016), AAAI, pp. 129–135.

[18] Borovies, D. A. Particle filter based tracking in a detection sparse discrete event simulation environment. Tech. rep., NAVAL POSTGRADUATE SCHOOL MONTEREY CA, 2007.

[19] Botea, A., Müller, M., and Schaeffer, J. Near optimal hierarchical path-finding. *Journal of game development 1*, 1 (2004), 7–28.

[20] Bourg, D. M., and Seemann, G. *AI for game developers.* " O'Reilly Media, Inc.", 2004.

[21] Brewer, D. Tactical pathfinding on a navmesh. In *Game AI Pro 360.* CRC Press, 2019, pp. 25–32.

[22] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. A survey of Monte-Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games 4*, 1 (March 2012), 1–43.

[23] Brusk, J., Lager, T., Hjalmarsson, A., and Wik, P. DEAL: Dialogue management in SCXML for believable game characters. In *Proceedings of the 2007 conference on Future Play* (2007), pp. 137–144.

[24] Campbell, J., and Verbrugge, C. Exploration in NetHack with secret discovery. *IEEE Transactions on Games 11*, 4 (2019), 363–373.

[25] Cenkner, A., Bulitko, V., Spetch, M., Legge, E., Anderson, C. G., and Brown, M. Passing a hide-and-seek third-person turing test. *IEEE Transactions on Computational Intelligence and AI in Games 6*, 1 (2013), 18–30.

[26] Čertickỳ, M., Churchill, D., Kim, K.-J., Čertickỳ, M., and Kelly, R. Starcraft ai competitions, bots, and tournament manager software. *IEEE Transactions on Games 11*, 3 (2018), 227–237.

[27] Chazelle, B. Triangulating a simple polygon in linear time. *Discrete & Computational Geometry 6*, 3 (1991), 485–524.

[28] Cheng, S.-W., Mencel, L., and Vigneron, A. A faster algorithm for computing straight skeletons. *ACM Transactions on Algorithms (TALG) 12*, 3 (2016), 1–21.

[29] Chin, W.-p., and Ntafos, S. Optimum watchman routes. In *Proceedings of the second annual symposium on Computational geometry* (1986), pp. 24–33.

[30] Chowdhury, M., and Verbrugge, C. Exhaustive exploration strategies for NPCs. In *Proceedings of the 1st International Joint Conference of DiGRA and FDG: 7th Workshop on Procedural Content Generation* (2016).

[31] Chvátal, V. A combinatorial theorem in plane geometry. *Combin. Theory Ser. B 18* (1975), 39–41.

[32] Craw, S. Manhattan distance. *Encyclopedia of Machine Learning and Data Mining* (2017), 790–791.

[33] Crecente, B. Volume is a pure stealth game with a slick aesthetic. `https://www.polygon.com/2015/6/25/8845833/volume`. Accessed: 2022-10-12.

[34] Critch, L., and Churchill, D. Sneak-attacks in starcraft using influence maps with heuristic search. In *2021 IEEE Conference on Games (CoG)* (2021), IEEE, pp. 1–8.

[35] Darken, C., and Anderegg, B. Particle filters and simulacra for more realistic opponent tracking. In *AI Game Programming Wisdom* (2008), vol. 4, Charles River Media, pp. 419–428.

[36] Demaine, E. D. Playing games with algorithms: Algorithmic combinatorial game theory. In *Mathematical Foundations of Computer Science 2001: 26th International Symposium, MFCS 2001 Mariánské Lázne, Czech Republic, August 27–31, 2001 Proceedings 26* (2001), Springer, pp. 18–33.

[37] Denisova, A., and Cairns, P. First person vs. third person perspective in digital games: do player preferences affect immersion? In *Proceedings of the 33rd annual ACM conference on human factors in computing systems* (2015), pp. 145–148.

[38] Díaz, J. M., and Verbrugge, C. Solving the take-down and body-hiding problems. In *Experimental AI in Games: An AIIDE 2019 Workshop* (Atlanta, Georgia, 2019), pp. 1–7.

[39] Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische mathematik 1*, 1 (1959), 269–271.

[40] Elfes, A. Sonar-based real-world mapping and navigation. *IEEE Journal on Robotics and Automation 3*, 3 (1987), 249–265.

[41] Elfes, A. Using occupancy grids for mobile robot perception and navigation. *Computer 22*, 6 (1989), 46–57.

[42] Falaki, P. M. M., Padman, A., Nair, V. G., and Guruprasad, K. Simultaneous exploration and coverage by a mobile robot. In *Control Instrumentation Systems*. Springer, 2020, pp. 33–41.

[43] Fikes, R. E., and Nilsson, N. J. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence 2*, 3-4 (1971), 189–208.

[44] Geraerts, R., and Schager, E. Stealth-based path planning using corridor maps. In *Computer Animation and Social Agents* (2010).

[45] Giesen, J., Miklos, B., Pauly, M., and Wormser, C. The scale axis transform. In *Proceedings of the twenty-fifth annual symposium on Computational geometry* (2009), pp. 106–115.

[46] Gill, A. Introduction to the theory of finite-state machines. *McGraw-Hill* (1962), 6–12.

[47] Graham, D. . An introduction to utility theory. In *Game AI Pro 360: Guide to Architecture*. CRC Press, 2014, pp. 67–80.

[48] Gregory, J. A context-aware character dialog system - game developer's conference. `https://www.gdcvault.com/play/1020386/A-Context-Aware-Character-Dialog`, 2014.

[49] Guerrero-Higueras, Á. M., Álvarez-Aparicio, C., Calvo Olivera, M. C., Rodríguez-Lera, F. J., Fernández-Llamas, C., Rico, F. M., and Matellán, V. Tracking people in a mobile robot from 2d lidar scans using full convolutional neural networks for security in cluttered environments. *Frontiers in neurorobotics 12* (2019), 85.

[50] Guy, R. K. *Combinatorial games*, vol. 43. American Mathematical Soc., 2000.

[51] Hagelback, J., and Johansson, S. J. Dealing with fog of war in a real time strategy game environment. In *2008 IEEE Symposium On Computational Intelligence and Games* (2008), pp. 55–62.

[52] Harabor, D. D., and Grastien, A. Online graph pruning for pathfinding on grid maps. In *Twenty-Fifth AAAI Conference on Artificial Intelligence* (2011), pp. 1114–1119.

[53] Hart, P., Nilsson, N., and Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics 4*, 2 (1968), 100–107.

[54] Hart, P. E., Nilsson, N. J., and Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics 4*, 2 (1968), 100–107.

[55] Hertel, S., and Mehlhorn, K. Fast triangulation of the plane with respect to simple polygons. *Information and control 64*, 1-3 (1985), 52–76.

[56] Hingston, P. A Turing test for computer game bots. *IEEE Transactions on Computational Intelligence and AI in Games 1*, 3 (2009), 169–186.

[57] Hladky, S., and Bulitko, V. An evaluation of models for predicting opponent positions in first-person shooter video games. In *2008 IEEE Symposium On Computational Intelligence and Games* (2008), IEEE, pp. 39–46.

[58] Hou, Q., Zhang, S., Chen, S., Nan, Z., and Zheng, N. Straight skeleton based automatic generation of hierarchical topological map in indoor environment. In *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)* (2021), IEEE, pp. 2229–2236.

[59] Huber, S., and Held, M. Computing straight skeletons of planar straight-line graphs based on motorcycle graphs. In *CCCG* (2010), Citeseer, pp. 187–190.

[60] Iida, H., Takeshita, N., and Yoshimura, J. A metric for entertainment of boardgames: its implication for evolution of chess variants. In *Entertainment Computing*. Springer, 2003, pp. 65–72.

[61] Innersloth. Among us, 2018.

[62] Isla, D. Probabilistic target tracking and search using occupancy maps. *AI Game Programming Wisdom 3* (2006), 379–388.

[63] Isla, D. Third Eye Crime: Building a stealth game around occupancy maps. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference* (2013).

[64] Jana, M., Vachhani, L., and Sinha, A. A deep reinforcement learning approach for multi-agent mobile robot patrolling. *International Journal of Intelligent Robotics and Applications 6*, 4 (2022), 724–745.

[65] Johansson, A., and Dell'Acqua, P. Knowledge-based probability maps for covert pathfinding. In *International Conference on Motion in Games* (2010), Springer, pp. 339–350.

[66] Johansson, M., Eladhari, M. P., and Verhagen, H. Complexity at the cost of control in game design. In *Proceedings of the 5th Annual International Conference on Computer Games and Allied Technology (CGAT 2012). Global Science & Technology Forum* (2012), Citeseer, pp. 22–29.

[67] Kacmarcik, G. Using natural language to manage NPC dialog. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (2006), vol. 2, pp. 115–117.

[68] Konami. Metal gear solid, 1998.

[69] Laguna, G. J., and Bhattacharya, S. Adaptive target tracking with a mixed team of static and mobile guards: deployment and activation strategies. *Autonomous Robots* (2019), 1–13.

[70] Lankoski, P., and Björk, S. Gameplay design patterns for believable non-player characters. In *DiGRA Conference* (2007), pp. 416–423.

[71] Lee, D.-T. Medial axis transformation of a planar shape. *IEEE Transactions on pattern analysis and machine intelligence*, 4 (1982), 363–369.

[72] Liu, Y., Chen, Z., Li, Y., Lu, M., Chen, C., and Zhang, X. Robot search path planning method based on prioritized deep reinforcement learning. *International Journal of Control, Automation and Systems 20*, 8 (2022), 2669–2680.

[73] Livingstone, D. Turing's test and believable AI in games. *Computers in Entertainment (CIE) 4*, 1 (2006), 6–es.

[74] Marzouqi, M., and Jarvis, R. A. Covert path planning for autonomous robot navigation in known environments. In *Proc. Australasian Conference on Robotics and Automation, Brisbane* (2003), pp. 1–10.

[75] McGee, K., and Abraham, A. T. Real-time team-mate ai in games: A definition, survey, & critique. In *proceedings of the Fifth International Conference on the Foundations of Digital Games* (2010), pp. 124–131.

[76] Meisters, G. H. Polygons have ears. *The American Mathematical Monthly 82*, 6 (1975), 648–651.

[77] Mendonça, M. R., Bernardino, H. S., and Neto, R. F. Stealthy path planning using navigation meshes. In *2015 Brazilian Conference on Intelligent Systems (BRACIS)* (2015), IEEE, pp. 31–36.

[78] Millington, I., and Funge, J. *Artificial intelligence for games*. CRC Press, 2018.

[79] Mononen, M. Simple stupid funnel algorithm. `http://digestingduck.blogspot.com/2010/03/simple-stupid-funnel-algorithm.html`. Accessed: 2019-07-20.

[80] Moravec, H., and Elfes, A. High resolution maps from wide angle sonar. In *Proceedings. 1985 IEEE International Conference on Robotics and Automation* (1985), vol. 2, pp. 116–121.

[81] Moravec, H. P. Sensor fusion in certainty grids for mobile robots. In *Sensor devices and systems for robotics*. Springer, 1989, pp. 253–276.

[82] Nau, D., Cao, Y., Lotem, A., and Munoz-Avila, H. Shop: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2* (1999), pp. 968–973.

[83] Nau, D. S., Au, T.-C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., and Yaman, F. Shop2: An htn planning system. *Journal of artificial intelligence research 20* (2003), 379–404.

[84] Niblack, C. W., Gibbons, P. B., and Capson, D. W. Generating skeletons and centerlines from the distance transform. *CVGIP: Graphical Models and image processing 54*, 5 (1992), 420–437.

[85] Nilsson, N. J. Shakey the robot. Tech. rep., SRI INTERNATIONAL MENLO PARK CA, 1984.

[86] Nowakowski, R. J. *Games of no chance*, vol. 29. Cambridge University Press, 1998.

[87] Ontanón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., and Preuss, M. A survey of real-time strategy game AI research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in games 5*, 4 (2013), 293–311.

[88] Orkin, J. Three states and a plan: the AI of fear. In *Game Developers Conference* (2006), vol. 2006, pp. 1–18.

[89] O'rourke, J. *Art gallery theorems and algorithms*, vol. 57. Oxford University Press Oxford, 1987.

[90] Overmars, M., Karamouzas, I., and Geraerts, R. Flexible path planning using corridor maps. In *European Symposium on Algorithms* (2008), Springer, pp. 1–12.

[91] Park, J.-H., Choi, J.-S., Kim, J., and Lee, B.-H. Roadmap-based stealth navigation for intercepting an invader. In *2009 IEEE International Conference on Robotics and Automation* (2009), IEEE, pp. 442–447.

[92] Paull, L., Seto, M., Leonard, J. J., and Li, H. Probabilistic cooperative mobile robot area coverage and its application to autonomous seabed mapping. *The International Journal of Robotics Research 37*, 1 (2018), 21–45.

[93] Pepels, T., Winands, M. H., and Lanctot, M. Real-time Monte-Carlo tree search in ms pac-man. *IEEE Transactions on Computational Intelligence and AI in games 6*, 3 (2014), 245–257.

[94] Redding, P. Aarf! arf arf arf: Talking to the player with barks. `https://www.gdcvault.com/play/1308/Aarf-Arf-Arf-Arf-Talking`, 2009.

[95] Riot Games. Valorant, 2020.

[96] Ruskin, E. AI-driven dynamic dialog through fuzzy pattern matching. empower your writers! - game developer's conference. `https://www.gdcvault.com/play/1015317/AI-driven-Dynamic-Dialog-through`, 2012.

[97] Russell, S., and Norvig, P. *Artificial intelligence a modern approach*. MA: Prentice Hall, 2009.

[98] Sabet, S. S., Griwodz, C., and Möller, S. Influence of primacy, recency and peak effects on the game experience questionnaire. In *Proceedings of the 11th ACM Workshop on Immersive Mixed and Virtual Environment Systems* (2019), pp. 22–27.

[99] Seiref, S., Jaffey, T., Lopatin, M., and Felner, A. Solving the watchman route problem on a grid with heuristic search. In *Proceedings of the International Conference on Automated Planning and Scheduling* (2020), vol. 30, pp. 249–257.

[100] Shute, G. M., Deneen, L. L., and Thomborson, C. D. An **O**(**nlogn**) plane-sweep algorithm for $\mathbf{L}_1$ and $\mathbf{L}_\infty$ delaunay triangulations. *Algorithmica 6*, 1-6 (1991), 207–221.

[101] Singh, N., and Verbrugge, C. Staying hidden: An analysis of hiding strategies in a 2d level with occlusions. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference* (2016), pp. 72–78.

[102] Snook, G. Simplified 3D movement and pathfinding using navigation meshes. In *Game Programming Gems*, M. DeLoura, Ed. Charles River Media, 2000, pp. 288–304.

[103] Soni, B., and Hingston, P. Bots trained to play like a human are more fun. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)* (2008), pp. 363–369.

[104] Straatman, R., Verweij, T., Champandard, A., Morcus, R., and Kleve, H. Hierarchical ai for multiplayer bots in killzone 3. In *Game AI Pro 360*. CRC Press, 2019, pp. 41–54.

[105] Studios, R. Batman: Arkham asylum, 2009.

[106] Sturtevant, N. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games 4*, 2 (2012), 144 – 148.

[107] Sugihara, K. Straight skeleton computation optimized for roof model generation. In *WSCG* (2019), vol. 27, pp. 101–109.

[108] Sukhatme, A. T. G. S., and Mataric, M. J. A multi-robot approach to stealthy navigation in the presence of an observer. In *IEEE International Conference on Robotics and Automation, New Orleans, LA* (2004), pp. 2379–2385.

[109] Sutiono, A. P., Purwarianti, A., and Iida, H. A mathematical model of game refinement. In *International Conference on Intelligent Technologies for Interactive Entertainment* (2014), Springer, pp. 148–151.

[110] Sutiono, A. P., Ramadan, R., Jarukasetporn, P., Takeuchi, J., Purwarianti, A., and Iida, H. A mathematical model of game refinement and its applications to sports games. *EAI Endorsed Transactions on Creative Technologies 2*, 5 (10 2015).

[111] Sutton, R. S., and Barto, A. G. *Introduction to reinforcement learning*, vol. 2. MIT press Cambridge, 1998.

[112] Tan, X. Fast computation of shortest watchman routes in simple polygons. *Information Processing Letters 77*, 1 (2001), 27–33.

[113] Tekinbas, K. S., and Zimmerman, E. *Rules of play: Game design fundamentals*. MIT press, 2003.

[114] Teng, Y., DeMenthon, D., and Davis, L. Stealth terrain navigation. *IEEE Transactions on Systems, Man, and Cybernetics 23*, 1 (1993), 96–110.

[115] Thrun, S. Learning metric-topological maps for indoor mobile robot navigation. *Artificial Intelligence 99*, 1 (1998), 21–71.

[116] Thrun, S. Particle filters in robotics. In *Uncertainty in artificial intelligence* (2002), vol. 2, pp. 511–518.

[117] Togelius, J. How to run a successful game-based AI competition. *IEEE Transactions on Computational Intelligence and AI in Games 8*, 1 (2014), 95–100.

[118] Tozour, P. Building a near-optimal navigation mesh. In *AI Game Programming Wisdom 1*. Charles River Media, 2002, pp. 171–185.

[119] Tremblay, J., Torres, P. A., Rikovitch, N., and Verbrugge, C. An exploration tool for predicting stealthy behaviour. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference* (2013), pp. 34–40.

[120] Tremblay, J., Torres, P. A., and Verbrugge, C. An algorithmic approach to analyzing combat and stealth games. In *2014 IEEE Conference on Computational Intelligence and Games (CIG)* (August 2014), pp. 1–8.

[121] Turing, A. M. Computing machinery and intelligence. In *Parsing the turing test*. Springer, 2009, pp. 23–65.

[122] Warpefelt, H., and Stråat, B. Breaking immersion by creating social unbelievabilty. In *Proceedings of AISB 2013 Convention. Social Coordination: Principles, Artefacts and Theories (SOCIAL. PATH)* (2013), pp. 92–100.

[123] Weber, R., Behr, K.-M., Tamborini, R., Ritterfeld, U., and Mathiak, K. What Do We Really Know about First-Person-Shooter Games? an Event-Related, High-Resolution Content Analysis. *Journal of Computer-Mediated Communication 14*, 4 (07 2009), 1016–1037.

[124] Wetzel, B., and Anderson, K. What you see is not what you get. In *Game AI Pro 3: Collected Wisdom of Game AI Professionals*. CRC Press, 2017, pp. 31–47.

[125] Widyaningrum, E., and Lindenbergh, R. C. Skeleton-based automatic road network extraction from an orthophoto colored point cloud. *The 40th Asian onference on Remote Sensing (ACRS 2019), Daejeon, Korea* (2019), 526–535.

[126] Woof, W., and Chen, K. Learning to play general video-games via an object embedding network. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)* (2018), IEEE, pp. 1–8.

[127] Xu, Q., Tremblay, J., and Verbrugge, C. Generative methods for guard and camera placement in stealth games. In *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference* (2014), pp. 87–93.

[128] Yannakakis, G. N., and Hallam, J. Capturing player enjoyment in computer games. In *Advanced Intelligent Paradigms in Computer Games*. Springer, 2007, pp. 175–201.

[129] Yannakakis, G. N., and Hallam, J. Towards optimizing entertainment in computer games. *Applied Artificial Intelligence 21*, 10 (2007), 933–971.

[130] Yannakakis, G. N., and Togelius, J. *Artificial intelligence and games*, vol. 2. Springer, 2018.