PATHFINDING IN DYNAMICALLY CHANGING STEALTH GAMES WITH DISTRACTIONS

by

Alexander Borodovski

School of Computer Science
McGill University, Montréal

April, 2016

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
MASTER OF SCIENCE

Abstract

Stealth games are genre of games that require the player to sneak past guards to reach a goal location. Designing a level for a stealth game is difficult, however, as the stealth paths in a level depend on a complex interplay of the physical space, the movements of enemies, and other factors. One such factor that makes the creation of good levels much more complicated, but also makes the levels much more interesting is *distractions*. They are player actions that change the movements of the enemies. By having dynamically changing guard movements, the realm of possible solutions increases greatly. The search space is thus also much larger, and hence finding good solutions becomes that much harder. Here, we create our own version of a probabilistic search algorithm to analyze a level and find stealth paths in the presence of distractions. This is done in a way that naturally allows variation in level design and enemy movements, while also allowing for a large variation in types and numbers of distractions that are present in the level for the player to use. We then introduce a series of optimizations that dramatically increase the success rate of the search while greatly decreasing its runtime, and create a compositional form of the search to solve larger, more complicated levels that are similar to real game levels. This design is integrated into the *Unity 3D* game development framework, which allows the creation and exploration of different levels, and exploration of how all of these factors including the placement and types of distractions affect the potential for stealth movement by players. We also model some levels from existing games to show the applicability of this design.

Résumé

Les *jeux d'infiltration* sont un genre de jeux vidéo dans lesquels le joueur a besoin d'éviter les ennemis et arriver à un lieu de but. Créer des niveau pour un jeu d'infiltration est difficile, parce-que les chemins dans lesquels le jouer peut tromper la vigilance des gardes dépend sur plusieurs facteurs de l'éspace, les mouvement des gardes, et comment ils se réagissent. Un de ces facteurs qui fait le création des niveaus beacoup plus compliqué, mais aussie les fait beaucoup plus intéressants est les *distractions*. Ils sont des actions de le joueur qui changent les mouvement des ennemis. Quand on a des mouvements des guards qui peuvent changer dynamiquement, il y a beaucoup plus de solutions pour les niveau. L'éspace de recherche est aussi plus grand, et donc, il est plus difficile de trouver des bons solutions. Ici, nous créeons notre propre version d'une algorithme de recherche probabiliste pour analyser des niveaus et trouver des solutions d'infiltration dans la présence des distractions. Nous le créeons dans un façon qui permet la variation dans la conçu des niveaus et mouvement des ennemis, et aussi permet beacoup de variation dans les types et quantités de distractions qui sont disponible pour le jouer d'utiliser. Après, nous introduissons des optimisations qui améliorent la vitesse et le taux de succès de notre algorithme et créeons une forme compositionnelle pour les niveaus plus grand et compliqués. C'est tout intégré dans *Unity 3D*, qui permet la création et exploration des différents niveau, et la exploration de comment tout ces facteur, comme les emplacement et types de distraction peuvent agir sur la potential de l'infiltration des joueurs. Nous avons aussi créer des modèles des vrais niveau pour démontrer la applicabilité de notre algorithme.

Acknowledgments

I would like to thank my supervisor Professor Clarke Verbrugge for all of his help in writing this thesis. I would also like to thank my Poochie, Hannah Belle Pomfret who has been the most supportive.

Contents

List of Figures

List of Algorithms

Chapter 1
Introduction

Stealth games are a popular genre of modern video games. In a pure stealth game level, the goal of the player is to reach the end of the level without being detected by the enemies (or guards). An example of such pure stealth games is the *Thief* series of games. In the simplest case the player has no control over guards behaviour and must simply study the patterns of guard movement in order to avoid being seen. However, in more complicated and more realistic games (or levels), the player can in fact affect the movements of the guards. One such interaction is through distractions. This involves the player doing something that causes the guards to investigate, changing their movement patterns and allowing the player to sneak by. There are numerous examples of what this distraction can be. In the *Splinter Cell* series of games the guards react to sounds, and the player can whistle to attract a guard, or throw something to cause guards to go somewhere else. With distractions the state space of the level becomes more complex and thus more interesting for the player, but also more difficult to model.

The complex interplay of many factors affects the existence and locations of possible solutions or stealth paths in a level, and tools that can find these paths in increasing complicated situations are of great use in level design and the creation of NPCs (Non-player characters) with stealth capabilities. Previous work in analyzing stealth games has considered only deterministic guard movements, with no player interaction [27]. Static definition of guard movements ensured that guard positions were known at all points in time. The inclusion of distraction effects means guard positions depend on when distractions are used by the player or stealthy NPC, causing the levels to be dynamic and greatly increasing the complexity of the search space. In this thesis we create a stealth level exploration tool that can handle distractions in an extremely flexible way which allows for the representation of a much broader portion of real game levels. This flexibility in turn means that the guard movements and lines of sight cannot be pre-computed, as they can change at any point during the search.

We first create a new way to represent the movements of guards such that they can be distracted at any point in time, and then implement an RRT (Rapidly exploring Random Trees) algorithm that searches the space and uses the distractions. We then construct numerous improvements to the RRT search allowing it to solve more complicated levels faster, and search the space in a much more efficient manner than a naive implementation. Following this, we construct a compositional version of the search to be able to solve very complicated levels that were composed by a combination of already non-trivial levels.

We first test the approach on a simple test level, and then we test a series of improvements on an artificially constructed level with much greater complexity. The greatly improved search is then tested on a few simple levels from existing games. Finally, the new compositional approach is shown to be effective on more complicated levels that were created which the initial search cannot solve.

The entirety of this design in implemented in the Unity3D game development framework. This shows that it is all easily implementable in and applicable to an industry standard game development framework, and that it can easily be used in conjunction with the game development process.

Specific Contributions include:

- Creating a new, dynamic model of stealth game levels with continuous space and discrete time that includes the use of distractions in a variety of ways.
- Implementing and significantly improving an RRT search that uses distraction and finds stealth paths in this new model.
- Creating a compositional version of the search for use in larger and more complex levels

- Testing and verification of results in a series of increasingly complicated test levels along with an existing game level.

This thesis contains four more chapters, the organization of which is as follows:

- Chapter 2 provides the background information and related work concerning the use of distractions and the use of RRT in exploring stealth game levels.
- Chapter 3 discusses the methodology, including new design of the levels and guards with distractions being an integral part of them, as well as the secondary methods used in the search such as triangulation of the space. Finally, it describes the search as a whole, including all of the improvements made and the compositional forms of the search as well.
- Chapter 4 discusses the levels modeled and designed to test the different searches and parameters within the search, as well as the results of the experiments using a variety of performance metrics.
- Chapter 5 gives a conclusion based on the results of the experiments, as well as possible extensions for future work.

Chapter 2
Background and Related Work

In this work, we discuss the use of an adaptation of the Rapidly-exploring Random Tree (RRT) search in order to find paths that are solutions to stealth game levels with distractions. Thus, in this chapter we first describe the necessary background information, about games in general and stealth games specifically, as well as the uses of RRT. We then continue to describe other relevant works concerning distractions, stealth games, RRTs and combinations thereof.

2.1 Background

In this section we describe the necessary background information for this work. We begin with an overview of stealth games and the game analysis tools that have been created in order to analyze them. Then we continue to a general description of the RRT algorithm and some of the fields in which it has been used.

2.1.1 Games Background

Games have been a field of research for computer scientists and mathematicians for years, with the focus being on mathematical analysis of combinatorial games. However, research has also been done in digital games of different genres. This work is concerned with analysis of digital games of the stealth genre. In this genre, the goal of the player is to make it to the end of the level without being seen by any enemies. In many modern games, stealth is just one component of the game itself, and purely stealth approaches are merely one of the possibilities for completing levels within them. However, pure stealth games do still exist and are still being created [5].

Stealth games present a challenge to the player to avoid all of the enemies. This challenge can be mitigated through level design and tools granted to the player to make accomplishing their goal easier. Environmental factors can include different lighting levels, obstacles and ambient noises which reduces the range of awareness of the guards, allowing the player to hide more easily. Additionally, the player may be given tools to either distract on incapacitate guards or teleport past them. An example of a map from a stealth game can be seen in 2.1, which from the game *Thief 2* [32]. It shows a floormap of a building with the locations of interest to the player being indicated. These include both where enemies (guards) are likely to be, as well as where items of interest may be. When designing a level, all of these things need to be taken into consideration to create a level of the appropriate difficulty, with a level being *stealth friendly* if tools that make sneaking easier are greater than the challenges of the level [25].

We are concerned with developing a tool to aid in stealth level design, specifically to help the designer understand the level they have created. Many such tools have been created to extract knowledge about games without using human players [18]. Often tools are created as part of the game design process and are specific to whatever game is currently being created and/or tested. However, there are also tools being created in game independent frameworks so that they can be reused in different games, and even different game genres and it is these tools for which there is a need [24].

Different, relatively game independent design tools have been created. These include one aimed at computing metrics on the optimal path for a player through a level assuming a given enemy and obstacle distribution [22], as well as a map-analyzing tool for the determination of strategic choke-points within a real time strategy game [19], and a geometric centrality and coverage method of analyzing map quality [21].

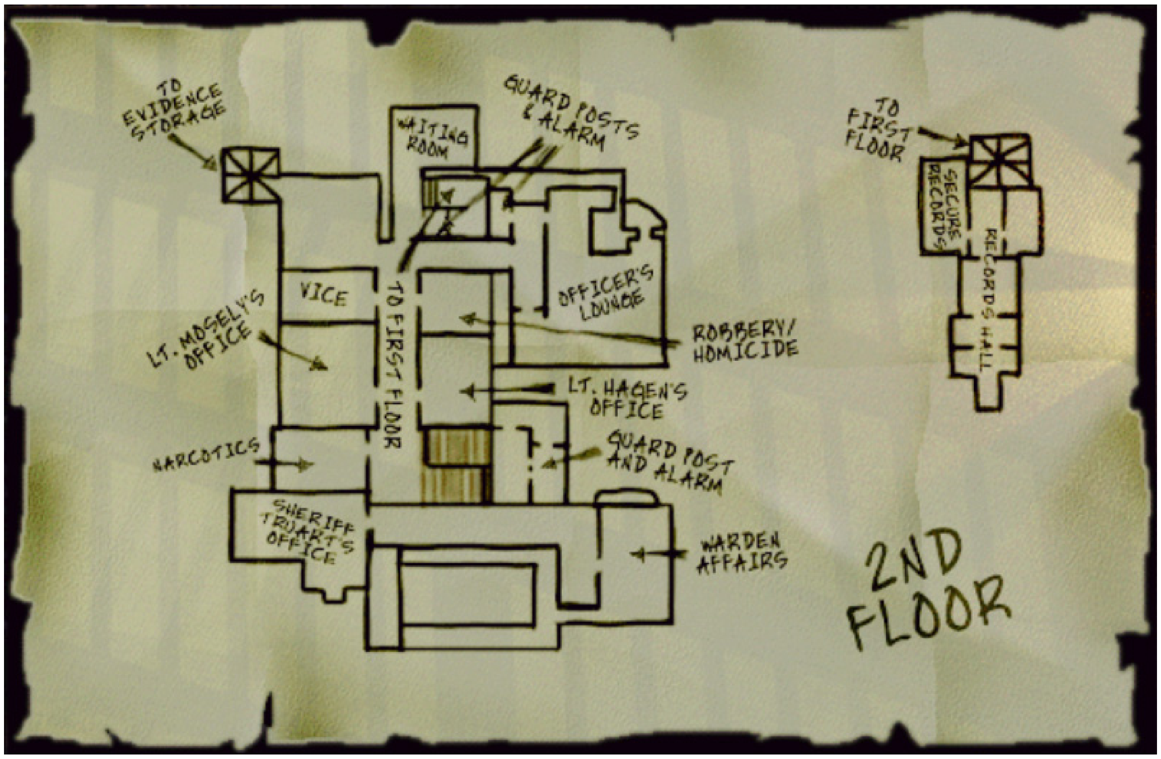


Figure 2.1 A Map of Level From Thief 2 [32]

2.1.2 RRT Background

The main search algorithm used in this thesis is Rapidly-exploring Random Tree (RRT) search. It is a random-based pathfinding algorithm commonly used in robotics for navigation of moving robots [17], and more recently used in games research [1]. There are also many other applications for the use of RRT including other robotics applications like controlling a high-degree-of-freedom robotic manipulator [23], and even in the field of bioinformatics for protein conformation changes [15]. It is probabilistically complete, so it will find a solution eventually, however in practice it benefits from being restarted. The search first explores the extents of the space in a sparse fashion before increasing the density of the search across the whole space and thus restarting causes the search to quickly choose a vastly different potential path through the space. In our case, it is used because of its random nature that allows it to model a great variation of different behaviour. This is always useful when attempting to model possible human behaviour. Additionally, it provides a cheap and flexible way to search the space. Other simpler algorithms such as A* are not suitable as we have an extremely large continuous search space and any direct search will take too long, as it will have too much space to search. In general RRT can be defined as follows: The input is an initial state at time zero, and a goal region, and the goal is to find a feasible path from the start to the goal region. At the beginning the only reached node is the starting node. At each step in the algorithm we randomly sample the space, and then attempt to connect the sampled point to the nearest point to it that we have reached so far. If the connection is feasible, the new point is connected and becomes part of the reached region. If the connection is not feasible the sample node is rejected. This search continues with randomly sampling nodes until the maximum number of nodes sampled is reached, or a path is found to the goal region. In general, to increase the likelihood that the goal region is reached, either the random sampling is biased towards the goal state, or after each node is connected, one can attempt to connect it to the end state.

RRT relies on an efficient mechanism for performing nearest neighbour queries. A convenient way to store the connected nodes and quickly find the nearest node(s) when new ones are being added is to use a KDTree [2].

2.2 Related Works

In this section we discuss some related works in different fields. We include a discussion of other work involving distractions, followed by work in stealth games both involving finding solutions and other game design for stealth games. Then, we continue to describe some work in the uses of RRTs outside of stealth games. This includes the application of RRTs to other games, as well as to other fields entirely.

2.2.1 Distractions

The use of distractions is well known in stealth games and is commonly used. It has come up as a common "trope" [30], and we have found a non-exhaustive list containing 66 entries of examples of game titles that use this mechanic [3]. However, as to the study of distractions the one example that was found was in a study on indexical story telling relating to how the locations of objects or remains can influence both the player and other non-player characters [8]. It mentions distractions as being story telling indices that other characters in the game can react to. However, no meaningful research was done about simulating any of this.

2.2.2 Stealth Games

Stealth Games using RRT

There has already been work in creating a search tool to find stealthy paths in game levels using RRT [27]. However in this case, the movements of the guards were fixed and had no reaction to player. This was in fact integral to the search process, since the space was voxelized with the pre-determined guard field of views being projected into the time dimension in order to pre-compute which locations are safe at what times. In this way, when performing the search the voxels along the path would simply be checked for whether they were occupied by guard view or obstacles.

That work was then extended to also include combat [28]. In that case the guards could then interact with the players via combat. However, this did not affect the movement model—combat occurred when a player path intersected a guard's field of view and resulted in the guard being disregarded in future states (as well as some player health loss). The guard movements remained deterministic, and so the pre-computed voxelized space remained viable. The work also involved the addition of resources to the level and their incorporation into the search process. Specifically, this resource was health packs, which would increase the health of the player after he had undergone combat and lost health. The way they incorporated them was by having a set percentage of the time that the node selected would be the node with the health pack. This was only true however for attaching to nodes where the player's health was not full. In this work we use a similar method for incorporating distractions into our search.

Stealth Games Not Using RRT

Other work in stealth games that did not attempt to find paths included a paper on the procedural placement of the guards [31], and one on the analysis of risk of paths that have been previously found [29]. They are both about game design, with the guard placement paper investigating the effect of different guard positions on the difficulty of the level as determined by using a stealth path-planning method, and the risk measurement paper investigates different risk metrics and evaluate them with a human study in order to get more information about level design from automatically generated paths.

There has also been research in creating artificial intelligence for the guards in stealth games in order to have their behaviour be more organic or believable. One such work is about the use of a variant of occupancy maps in order to have the enemies track the position of the player using a probability distribution that becomes more diffuse the longer a player remains out of sight of the enemy [12].

Another application of finding solutions to game levels is as an authoring tool. There is a work in which planning techniques are used to find a solution to a game level based on the gameplay constraints, and then the solution found is displayed as a storyboard [20]. Their prototype did this for the game *Hitman* which does involve elements of stealth in addition to combat.

2.2.3 RRT

RRT in Non-Stealth Games

The RRT path-finding method was also applied to a different game genre, that of platformers [26]. In that genre, there is a physics based movement system, and so connectivity between two nodes in the RRT was more complicated to determine. Instead of just performing checks for collisions with obstacles and enemies, a motion planner was required to try to search for a way to connect the nodes by using the range of movements available to the player. This work was one that used a distance measure to only sample nearby points with the RRT. Additionally, several methods were used to try to bias the search. This included dividing the spatial search space into a grid and preventing the sampling of grid nodes that were deemed ‘oversampled’ in order to try to push the search out to cover more space instead of searching all of time within a small physical space.

RRT in Non-Game Settings

There have been numerous works involved in improving the RRT algorithm in different ways. One such work involved improving the search by incrementally building two RRTs, one from the start and one from the end in order to more quickly find a solution [16]. However, this method is not applicable to our case due to the interactions of the player with distractions. We cannot search from the end as we do not know what changes to the state of the level were made earlier in the path, and these kinds of changes are required to find a solution.

There was also work towards making RRT work better in the presence of narrow passages by the use of a retraction-based RRT planner [34]. This involves taking sample points that are within obstacles and pushing them out to the edge of the obstacle so that the search can attempt to connect them to the search tree. Instead of this, in our search we use a method to completely prevent sampling of obstacles.

As RRT is a heuristic exploration algorithm, it does not find optimal paths. This is in contrast to another popular path-finding algorithm A-Star [33]. Due to this, there was work in trying to make an RRT search that would converge to the optimal path, and it was called RRT* [14]. However, it was proven that convergence for that algorithm could take arbitrarily long. A different work created a fast-converging version of RRT* called Rrt*-smart [13] which would quickly converge to an optimal path. However, in our case our goal is not to find an optimal path, but to explore the space and find many different paths. Thus, these improvements are not of use for our purposes.

Chapter 3
Methodology

In this chapter we describe how we define levels, and all that is involved in the process of finding a solution to these levels. It is divided into three sections. The first is level definitions and contains descriptions of how we define levels and the things they contain such as distractions and guards. The next section, RRT Details, contains the description of important details specific to our implementation of RRT. Then, there is a section outlining our process for triangulation, which is required for many of the improvements we make to the search. These triangulations are done to have a good way to represent the unoccluded space, as the obstacles can be arbitrarily placed polygons anywhere in the space. Additionally, this gives a good basis for calculating distances along geometrically feasible paths, since game levels often have a dense maze-like structure and simple Euclidean calculations are a poor approximation of distance in them. The final section details the process of the search itself. It also contains details of improvements we have made to the search, the use of multiple searches and the compositional version of the search.

3.1 Level Definitions

As the purpose of the player is to reach the end of the level without being seen by guards, while possibly using distractions, we use this section to first describe what a level is composed of. We then continue with an overview of how distractions are defined, and conclude with a description of how the guards are defined and how distractions act upon the guards.

3.1.1 State Space

The levels are continuous two-dimensional in space, and have a discrete, positive time dimension. Thus, the domain can be defined as a patch of two-dimensional Euclidean space extruded over time which is positive discrete, denoted by $\Sigma \subseteq \mathbb{R}^2 \times \mathbb{N}$. There is a player whose goal is to reach the end, however there are also guards, or enemies, who patrol the level which the player must avoid. The levels contain obstacles, through which the player cannot pass, and through which the guards cannot see or pass. Thus, we can define the free space as $\Sigma_{free} = \Sigma - \Sigma_{obs}$, where Σ_{obs} represents the space taken up by the obstacles, similarly to Tremblay [28]. The player loses if they enter the field of view of any of the guards. The field of view of the guards is in a cone in front of them with an angular width of 66 degrees, and a length of five units. These values were chosen to approximate reasonable fields of view in games, and can be changed if desired. The guards have set patrol routes that they will follow in a loop. We will denote the set of guards as E. Additionally, the level contains one or more distraction points, the set of which will be denoted as D. Thus, at this point our game state can be denoted as $\Sigma \times E \times D$. In this context, a path is defined by a series of points in three-dimensional space with monotonically increasing time values, also known as a trajectory. The movement between the key points is to be linearly interpolated. A feasible path is one that respects the movement constraints of the player, level and enemies. In this case, the player has a set maximum speed, cannot move through obstacles or enter the field of view of any guards. Then, a successful path is one that goes from the start to the end and is feasible.

3.1.2 Distraction Points

Distractions come in many forms. In this case, in order to cover as many cases as possible distractions are defined in the following manner. Each distraction has a point in space that the player needs to reach in order to use it. Once the player reaches the distraction point, a signal is sent out to all guards that that specific distraction point was activated at that specific time. A distraction event can be denoted as $T_D = \mathbb{N} \times \mathbb{N}$ as a distraction time composed of two numbers, the time and the index of the distraction. Then, certain guards, depending on their current location and pre-set patrol routes will react to the distraction by changing where they are going, and entering a new looped patrol.

This new patrol may in fact be them going to check a point and then resuming their prior pattern, or changing to a new pattern. The location that the guards first go to, when the distraction is activated may be where the player activated it, or it may be in a different location. Additionally, a distraction can be set up to be time delayed, by having the signal have a future time instead of the time the player actually visited the distraction point. In table 3.1, we can see eight variations on distractions that can thus be implemented. This is based on the location the guards (Distraction Event Location) go to being the same as or different from the location the player goes to activate the distraction (Distraction Activation Location), as well as by the time of activation versus time of the event. A third measure that is used is whether the path the guard patrols after the distraction is the same as before the distraction or not. This measure is the least distinct in the context of real games, because most games will eventually reset to the original paths, but it may be after having the guards follow different paths for an extended period of time. There can be any number of different, independent distraction points each affecting different (although perhaps overlapping) sets of guards, or affecting the same sets of guards in different ways.

Many different games have examples of different variations from table 3.1. For example, variations 1 and 5 are commonly seen and occur in games such as the *Splinter Cell* series and the *Fallout* series. In *Splinter Cell* it is commonly seen as broken glass on the ground which makes a noise if you step on it, causing guards to investigate. In *Fallout 4*, there are sometimes chains of tin cans hanging from the ceiling, and upon walking into them, they make noise which the enemies come investigate. Another example is of variation 2 or 4 in the game *Far Cry 4*. In this game you can throw a rock, which will make a noise, and cause nearby guards to investigate, before returning to their previous position. This can either be seen as variation 2, because as soon as the rock lands, the guard goes to investigate, or as variation 4 because of the time between the player throwing the rock and it landing. There are lots of other examples in different games that are not discussed here.

Table 3.1 Possible Distraction Combinations

Possible Different Combinations Locations,Times and Paths								
Variation Number	1	2	3	4	5	6	7	8
Distraction Activation Location	A	A	A	A	A	A	A	A
Distraction Event Location	A	B	A	B	A	B	A	B
Distraction Activation Time	T1	T1	T1	T1	T1	T1	T1	T1
Distraction Event Time	T1	T1	T2	T2	T1	T1	T2	T2
Old Path	P1	P1	P1	P1	P1	P1	P1	P1
New Path	P1	P1	P1	P1	P2	P2	P2	P2

3.1.3 Guards

Individual games use various, proprietary methods for controlling NPCs. For guard patrol behaviours we will assume a specific model for implementing guard behaviours based on a waypoint system, a common approach to defining fixed behaviours in game design. There are three main types of waypoints corresponding to the three basic actions that a guard can do. These are waiting, rotating and moving. The waypoints store the speed (of moving or rotating) or the duration (for wait waypoints). Additionally, each waypoint has an indication of which waypoint is next, i.e. what to do next. In this way, the waypoints define the patrol patterns of the guards. At the beginning, each guard has an initial position, orientation and first waypoint. Once they reach the waypoint they follow the links forever.

An important property of our model is that the waypoints also control the distraction behaviour. Each waypoint has, in addition to next waypoint, a list of distraction waypoints; one for each distraction in the level. When the distraction is activated they stop wherever they are, rotate to face the distraction point and move to it. Once they have reached it they continue following waypoints as normal beginning with the waypoint indicated by the distraction waypoint. Once a guard has been distracted, his current waypoint is the distraction waypoint. Much like all the others, it too has a list of distraction waypoints which dictate the actions of the guard if a second distraction is activated after the one that sent him to this distraction. It can be configured such that the guard will ignore additional distractions, or that the guard will branch off to the new distraction.

For the purposes of the search, each guard has a method which takes as input the desired time, as well as the times of distractions if any, and returns the guards position and orientation at that time. The way this is done, is that starting at time zero, the guards movement is simulated until the desired time, taking into account the distractions. In order to speed this up, we do not simulate the guards moving one time unit at a time, but instead we calculate the time the next waypoint would be reached and jump to that time. For distractions, we calculate position until the distraction, and then calculate as normal from there with the different goal waypoint.

An example of the guard paths for a level can be seen in figure 3.1. The white space is Σ_{free}, while the black space is Σ_{obs}. The guards are represented by the small yellow spheres, with the orange lines being indications of their field of view. The guards initially patrol by cycling counter-clockwise around the central obstacle as denoted by the blue arrows. Then, if a distraction occurs, for any guard whose closest corridor is the bottom left one, they move to the distraction point there as indicated by the red arrows. Similarly, any guards for whom the closest corridor is the top right one, move to that distraction point. Afterwards, the guards return to the nearest corner of the old square patrol route, and continue patrolling as normal, as indicated by the green arrows.

An example of the waypoint structure for that level can bee seen in figure 3.2. Here, we have the waypoints set up in approximately the same relative positions as in the level itself. However, each waypoint is represented by its name and the two pointers it has, which are the links for next waypoint, and distraction waypoint that are required for the behaviour described previously. The link to next waypoint is shown in blue, while the link to the distraction point is shown in red. We use WP for waypoint, and WP D for distraction waypoint. It can be seen that those waypoints that indicate places where the distraction should not occur have the distraction waypoint pointing to the same place as the next waypoint so the reaction to the distraction event will be to continue as before. This diagram is simplified in that it does not contain the rotation waypoints that are located in the same places as some of the waypoints indicated here and control the rotation of the guards. Additionally, for more distractions there would just be longer list of pointers to different distraction points, one for each distraction in the level.

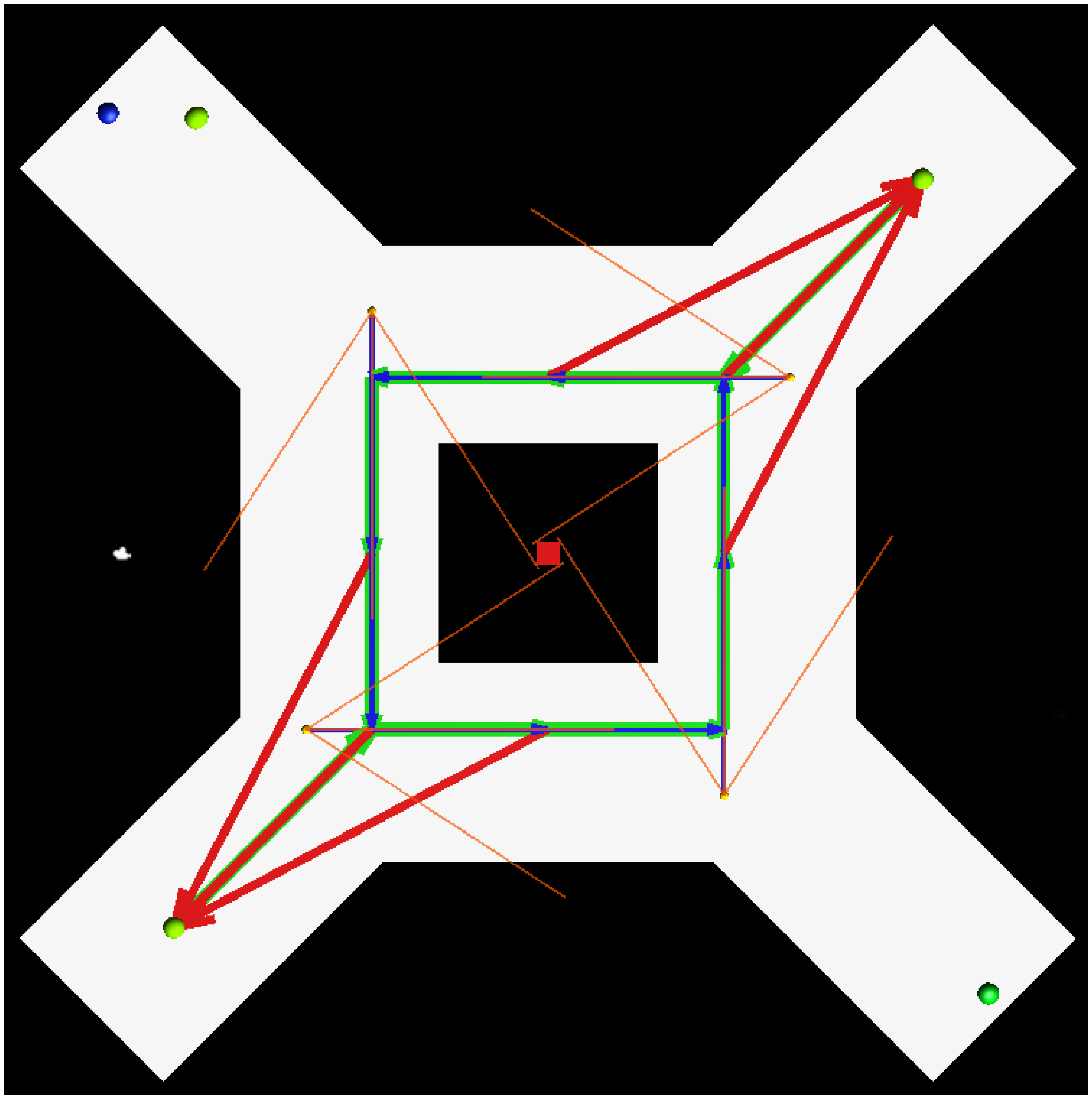


Figure 3.1 Guard Movements for the Multi-Alarm Level. Blue Arrows indicate pre-distraction movements, red arrows indicate movement resulting from distraction, and green arrows indicate post-distraction movements.

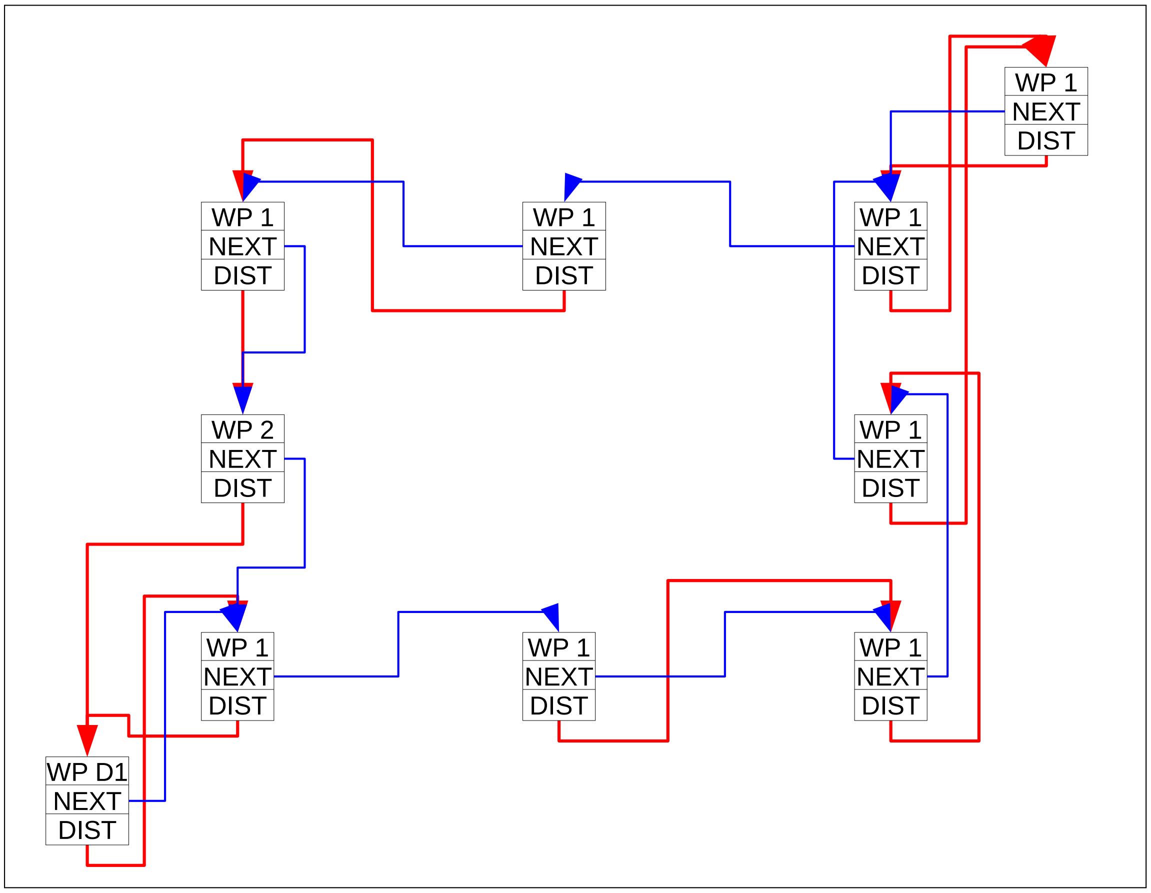


Figure 3.2 Waypoint Structure of Mult-Alarm Level

3.2 RRT Details

In this section we outline important details of our implementation of RRT. This includes the state that we store in each node, as well as how connectivity is determined when adding new nodes to the tree.

3.2.1 State Within RRT Nodes

When constructing the RRT, each node needs to contain all of the relevant game state. In this case it is includes the position and time of the node, the list of all guards, as well as the time each distraction (if any) occurred. This can be represented as $\langle x, y, t, E, d_i, d_t \rangle \in$

$\Sigma \times E \times T_D$. d_i and d_t can be extended to lists for multiple distractions, and this is in fact what is done in our implementation. It can also be noted that in our implementation, E, the set of enemies, does not ever change, but if this were, for example, extended to include combat like in Tremblay's work [28], then it could.

3.2.2 Connectivity Check

When using an RRT to search a space, we need to compute whether or not two points can be connected. In this context, we need to check four things to do this: check if we have visited, if the maximum speed constraint is respected, if there are collisions with obstacles, and if the field of view of any guard is intersected.

We first check if the node is a node we have already visited, as visiting the same node multiple times is nonsensical. Next, we check if the player could reach this node from the previous one in time if he was moving at maximum speed. This is done by calculating the angle in 3D space (where the third dimensions is time).

Then, we need to know if the line segment intersects with the physical obstacles in the level. To do this, we constructed our obstacles using objects in Unity 3D that have colliders, and checked for collision using the built-in Linecast in Unity.

We also need to know if the line segment will cause the player to come into the field of view of any of the guards. Computing this directly is awkward, as to use Unity's Linecast we would need to generate a complex shape based on the way a field of view moves, rotates, and is affected by occlusion. Instead, and as time is discrete, we compute this heuristically using a binary search over the time component, going down until we check every 3rd frame. For a specific frame we first determine the position and orientation of the guard. Then we check if the player is close enough to the guard to be seen based solely on distance between them. If the player is close enough, we check if the player is within the field of view of the guard based on angle alone. If the player is, we simply do another Linecast to see if there are obstacles in the way. If not, then the guard sees the player.

At this point, if all of the checks have been passed, then we have found a connection that the player can reach without being detected by the guards.

3.3 Triangulation

Some of our improvements to the basic RRT rely on having a spatial decomposition of the level geometry. In this section, we thus outline our methods for triangulation, and other triangulation related computations. This is because we need to compute a triangulation, as well as a DAG structure for it, distance metrics, and simple paths to the end in order to implement a number of improvements and advancements to the search, which are outlined in the next section, section 3.4.

3.3.1 Triangulating

The level geometry is defined by a floor and a series of possibly overlapping obstacles. All of these are rectangles. The floor represents the space that could be passable. Any part of the floor that overlaps with one or more obstacles in not passable. So, we first iterate through the list of all obstacles, and any overlapping ones are merged into a single polygon. At this point, we separate the resulting polygons into those that are fully contained within the floor rectangle, and those that overlap with the floor rectangle. The polygons that overlap with the boundary are merged with it to create a new boundary. In this way, we get a boundary that contains some polygons of unreachable space. From the vertices of these shapes we get the list of all vertices that will be used for the triangulation. Let the list of obstacles be known as O, and the list of vertices be known as V.

The method used to compute the triangulation can be seen in algorithm 1, on page 35. To compute the triangulation, we iterate through every pair of vertices and if the line between them does not intersect any obstacles, add it to the list of lines in the triangulation. Then to form triangles, iterate through the list of lines in a triple loop, looking for sets of lines that form triangles. Every unique triangle is then added to the list of triangles. Once we have the list of triangles, we determine triangle adjacency as shown in algorithm 2 on page 36. In it, we go through every pair of triangles and determine whether they share an edge. Thus, we determine which triangles are connected to each other, and through which shared edge.

We desire a *Delaunay* triangulation as this improves triangle shape, reducing the chance

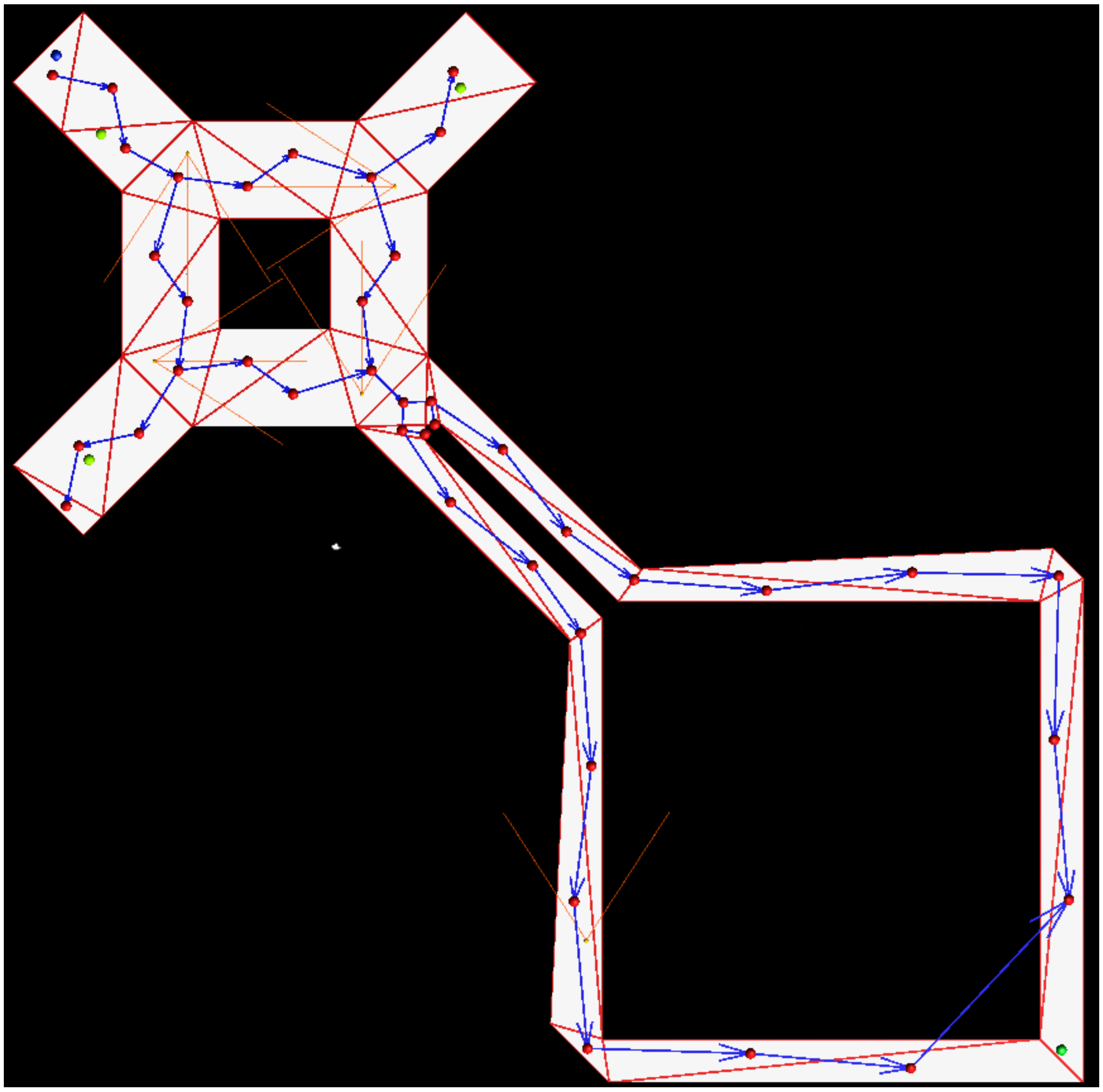


Figure 3.3 The DAG Structure Drawn for the Two Part Choice Level

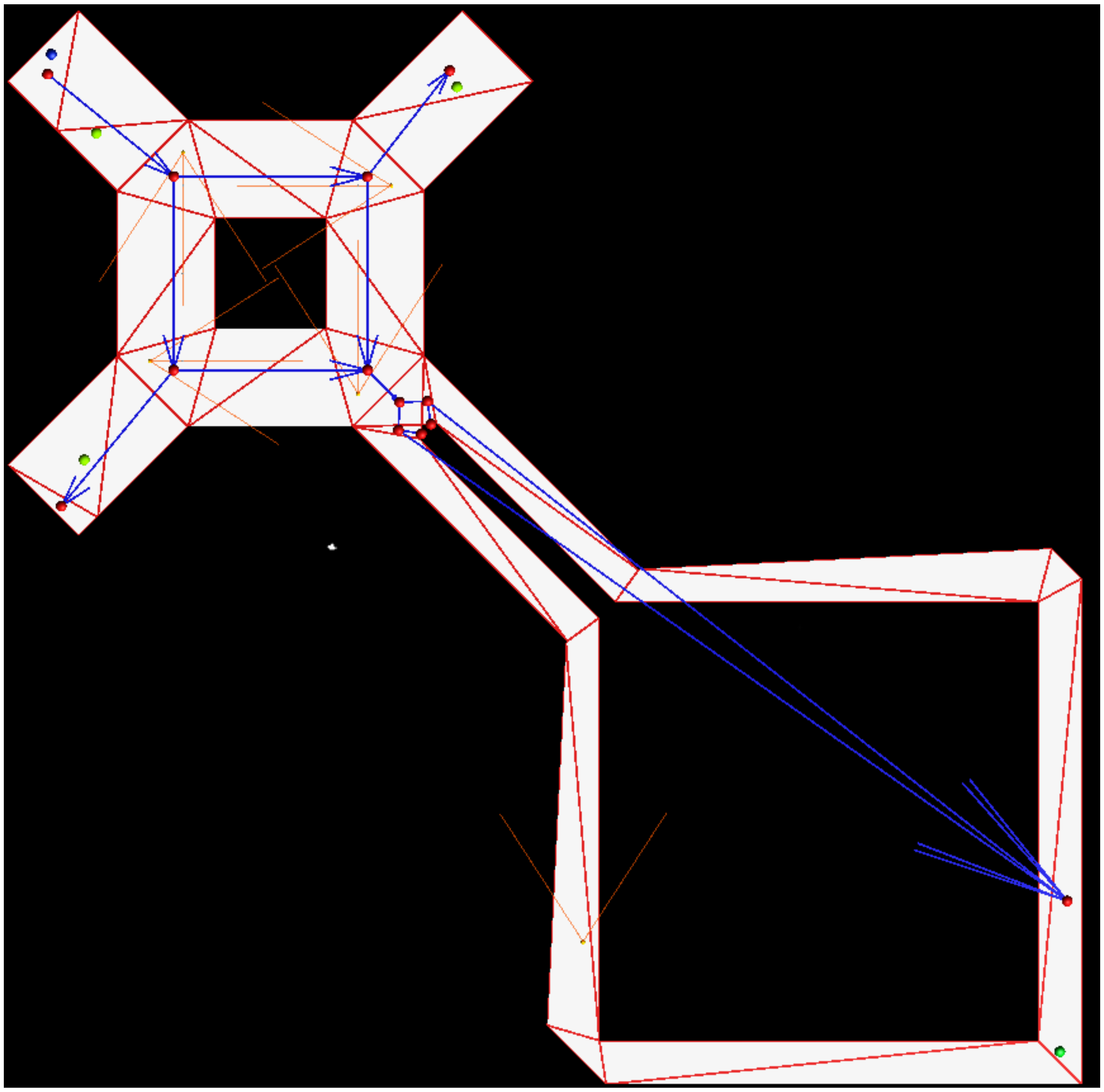


Figure 3.4 The Simplified DAG Structure Drawn for the Two Part Choice Level

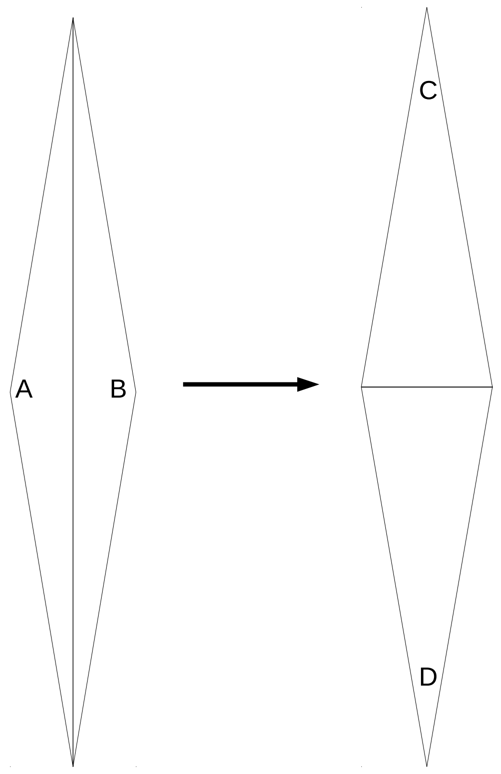


Figure 3.5 Edge-Flipping Example: Note that angles A and B sum to more than 180 degrees, while angles C and D sum to less.

of numerical issues due to overly skinny triangles, and also reducing the lengths of edges. So, we determine if the triangulation is a Delaunay triangulation [7], and make it Delaunay if it is not by using algorithm 3 on page 37. We do this by iterating through all of the edges that are shared between triangles. For each edge we check if the sum of angles forming the edge are greater than 180 degrees. If so, we flip the edge and recompute adjacencies for the newly created triangles. Once we loop through all of the triangles, we have a Delaunay triangulation. This is an algorithm that is known to converge [6]. A graphical illustration of edge-flipping can be seen in figure 3.5. It is also worth noting that this will be a *constrained* Delaunay triangulation as we can only flip edges that are shared by two triangles if they are interior to the geometry. Edges that are part of obstacle or level boundaries cannot be changed, which means that at some points the Delaunay property can be broken. This does not affect convergence of the edge-flipping algorithm, and since we use the Delaunay property as a heuristic improvement it does not change correctness in our approach. An example of a triangulated level can be seen in figure 3.9. The free space, in white, is subdivided into a series of red triangles.

It can be noted that we use naive, brute-force methods for the computation of our triangulation. However, as the total analysis times are dominated by the search, and adding nodes, the difference in time given by a more efficient triangulation algorithm is not taken to be significant, and thus the brute-force methods are sufficient for our purposes.

3.3.2 Triangulation Distance and Paths

One of our improvements to search (in subsection 3.4.2) will require we calculate a distance measure from the starting triangle. For this we construct a DAG (directed a-cyclic graph) structure for the triangulation. At first, we have a list of all of the triangles, and a graph structure based on adjacency since every triangle has references to all triangles with a shared edge. To construct this DAG we set the triangle containing the start node as the root of the DAG. Then, we perform a depth first search of adjacency structure to calculate depth. In the simple case, we simply have depth increased by one every time we visit an adjacent triangle. However, adjacency structure can have numerous loops, so that is only the case when we visit unvisited triangles. If we visit a triangle that has been visited before we compare the depth of the triangle being visited to the depth of the triangle we are coming from. If the depth is greater than one plus our current depth, we set the depth to that value and re-calculate the depth of all of its adjacent triangles, since we have found a shorter path to it. Otherwise, its currently calculated depth is correct, and we do not need to visit it again.

In addition to calculating depth, we create the DAG structure, such that such that for any given triangle its children are adjacent triangles with greater depth than it, and its parents are adjacent triangles with less depth than it.

An example of the DAG structure of a level can be seen in figure 3.3. Here, the triangulation is also drawn. The red spheres represent the centers of the triangles, with the blue arrows pointing from parents to children in the DAG structure. It is important to note that these arrows are only indicating the DAG structure and not anything relating to pathing, so their penetration of the obstacles is acceptable.

An additional adjustment was made to ensure that the triangle containing the end node would always be a leaf. This was done by making the triangle containing the end node impassable to the search. Thus, any triangles adjacent to it that are reachable from the start without passing through the end node will be considered its parents, and it will have no children.

For a future improvement, described in subsection 3.4.4, called the multiple midpoint method it was necessary to find a way to calculate all simple paths within the level. This was done by simplifying the triangulation DAG. Essentially all nodes (triangles) that only have one parent and one child were removed, linking the parent to its grandchild directly. Thus the DAG retains the start node, all leaves, and nodes that are splits or merges (having two or more children, or two or more parents). As the triangulation is a direct representation of the open space in the level, this simplified DAG structure represents all possible simple paths through the level.

An example of a simplified DAG structure of a level can be seen in figure 3.4. Much like in the figure for the normal DAG structure, the triangulation is shown, and blue arrows indicate the connections of parents to children. It can be seen that each node (triangle), represented by a red sphere is one of the start, a leaf, a node with two or more parents, or a node with two or more children.

To calculate simple paths, we make a separate path for every possible combination of choices that lead to a leaf in the DAG. Then, if we want end paths, we simple take only those paths that end in the leaf containing the end node.

Once we have these end paths, calculating distance along a certain path is easy. We define a distance metric between two triangles in subsection 3.4.2, and using it, we simply visit the triangles along the DAG, referencing the path whenever we have a choice of which children to visit until we reach the desired triangle. Thus, we can calculate the distance along a path to any triangle in that path from the path's beginning.

3.4 The Search

In this section we outline the search process as well as all the variations therein. We begin with a description of the basic search that we used, including how we integrated distractions. Then, we continue with issues we found with the search and improvements that we made. Next, we have motivation for using multiple small searches instead of one big search. We conclude with a description of compositional forms of the search.

3.4.1 The Search Process

For the purposes of the RRT search, our search tree is stored as a KDTree in three dimensions; the x coordinate, the y coordinate and the time, $\langle x, y, t \rangle$. Thus, when the nearest node is queried from the KDTree, these are the dimensions in which nearness is calculated. We randomly sample in all three dimensions to find possible nodes to add to the tree. Every time a node is added, we check if it is at the location of our goal, which would mean our search succeeded. We also try to connect it to the goal node, as it is unlikely that we would reach the exact position of the goal by choosing random points in space. The time used for this connection is the time of arrival at the end assuming moving at maximum speed to it.

To deal with distractions a certain percentage of the time, in our case 20%, instead of sampling the space we pick the location of a random distraction point at a random time and try to connect to the tree. 20% was chosen based on a small number of runs as a good percentage, but it can be changed if we want more or less bias to the distractions points. If the connection in fact happens, then at that node the distraction happens at the time the distraction point was reached. Thus, any further nodes that connect to that node will have the distraction as activated in their state. This means that when a distraction is reached it must be activated. This is done to ensure that there are sufficient states that actually include the activation of a distraction. The option of approaching the same point without using the distraction is covered by having the search be able to sample the same point (or an arbitrarily close one) in its random sampling step which allows the exploration of the space without distractions. The ratio of nodes where a distraction is used, and where a distraction is not used can be controlled in an indirect manner by changing the percentage of the time that the distraction nodes are sampled.

3.4.2 Issues and Improvements

This subsection details issues found and improvement made to the search. It begins with a description of a series of simple improvements, followed by a description of an unexpected problem which we dubbed *distraction stacks* and how they were eliminated. It finishes with a section on triangulation-related improvements to the search.

Simple Improvements

In order to increase the speed and/or success rate of the search three of improvements were made. One simple improvement was to pick x and y values first, and then select only such time values that the point would be reachable by moving towards it at maximum speed from the start. This removes a small but not insignificant piece of the sampling space leading to more useful nodes being selected. This can be seen illustrated for a two-dimensional space (distance and time) in figure 3.6. Here the red region represents area that requires greater than the maximum velocity to enter from the origin, and which we therefore never need to sample, irrespective of the speed required to get to a sample point from a given nearest neighbour in the tree.

Early experiments also showed that Linecasts were a time-consuming computation. Our method to speed up the search in this respect was to shift the Linecast cost to a pre-computation step: we discretize the space, and then precompute the Linecast from every point to every other point in the level. Although this in and of itself is time-consuming, it only needs to be computed once per level and then multiple searches can be done on the same level. Also, if the space is divided into a sufficiently large number of grid cells, the discretization should not have any bearing on the accuracy of the results.

We also observed that in our initial design that samples were more likely to be rejected if they were far away from any nearest neighbour in the tree. Nodes that are very far away are less likely to connect to the tree, due to the larger number of obstacles that tend to intersect a straight-line connection. Our next improvement was thus to introduce a distance constraint on sampling. This is defined by a three-dimensional bounding box for bounds

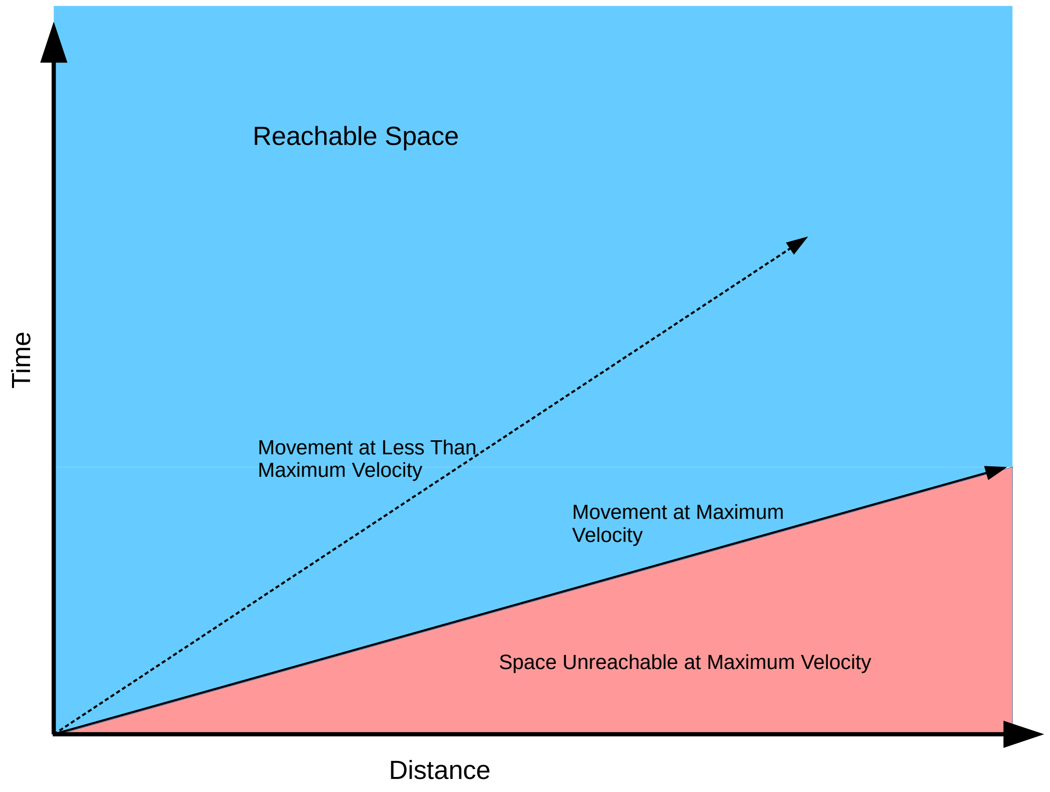

Reachable Space
Time
Movement at Less Than Maximum Velocity
Movement at Maximum Velocity
Space Unreachable at Maximum Velocity
Distance

Figure 3.6 Space Unreachable at Maximum Velocity

in x, y and t. It bounds the tree with an offest of distance, d. Each time a node is added, we update the bounds of the box. This is done by separately computing the maximum of current bound and the value of the added node plus d in each direction. In this way, the bounding box is updated in each direction independently. By doing this, instead of searching the entire space, we would only search the space within a certain distance of the nodes that had already been added to the tree. This leads to more sample nodes being successfully added to tree.

Distraction Stacks

After some prototyping, a problem was found with the nodes located at the distraction points. A distraction point can be seen as a point in x, y denoted as $\langle d_x, d_y \rangle$, at which a series of possible tree nodes can be located, each represented as $\langle d_x, d_y, t \rangle$ for a variety of values of t. Since the distraction points were selected rather often, due to our bias, the RRT trees would begin to take an unfortunate shape. This can be seen in figure 3.7, wherein we are given an angled view of the level with the grey tree structure coming up from the level. Once the RRT successfully reaches the distraction point, any other points located there have a tendency to find previous points in the same location as nearest neighbours. The search tree thus tends to build branches which are stacks of attempts to use a distraction, only the first of which is effective. In essence the tree search is modeling the equivalent of causing a given distraction, and then staying there for a variety of different amounts of time. Remaining at a distraction point can sometimes be important of course, but a more generally useful search behaviour is to have the tree activate the distraction at a variety of different times, not just linger there for different durations.

In order to fix this issue, when a node is being added at the same coordinates as a distraction point, it does not just try to connect to the nearest node of the RRT tree. Instead, it finds the three closest nodes, and looks for one that is in a different location. It looks for a node $\langle x, y, t \rangle$, where $x \neq d_x$ or $y \neq d_y$. If none are found, then the node is rejected and not added. This prevents the player from lingering at a distraction point after they have used the distraction point. Remaining nearby is still possible, however, as they can remain arbitrarily close if the random sampling chooses nearby points. An example of a search with this fix in place can be seen in figure 3.8. It can be seen that the tree is much more spread out, and has searched the space more effectively.

Triangulation-Related Improvements

Another issue that was found occurs due to the dense, maze-like nature of many game maps. In these environments a large portion of the space of the level can be taken up by obstacles. Any points that are within the obstacles do not need to be sampled as they are inherently unreachable. So, we triangulated the space between the obstacles, so we could search

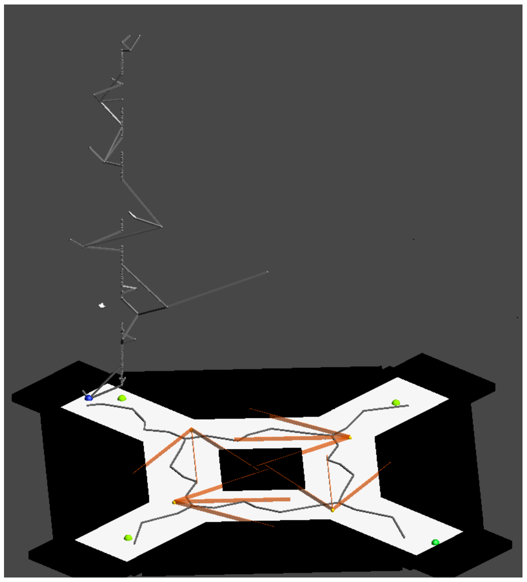


Figure 3.7 An Example Search with Distraction Stacks

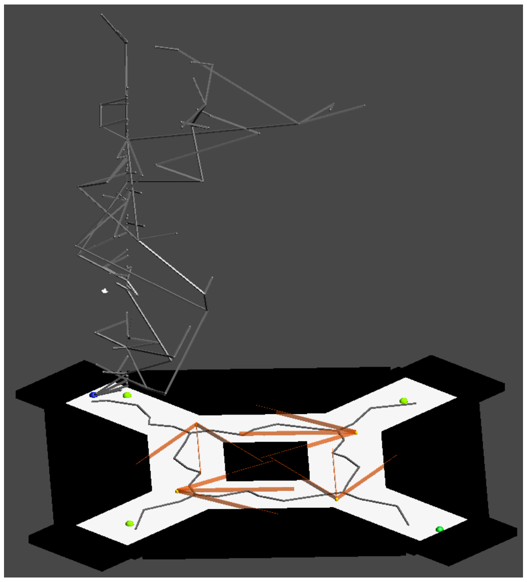


Figure 3.8 An Example Search with Distraction Stack Prevention

only obstacle-free space by sampling the triangles. A Delaunay triangulation was used for multiple reasons. In games they are often preferred due to the way they (heuristically) reduce the number of long, thin triangles, which tend to be sources of numerical error, and which may also be too thin to accommodate the graphical representation of a character. An interesting further reason specific to our search design, is that with a Delaunay triangulation less of the space will be taken up by edges. Although conceptually infinitely thin, when we sample points from a triangle we include points that lie on the triangle boundary, and so any shared edges are more likely to be sampled than the space within a triangle. Having less of the space being taken up by edges means that the space will be more evenly sampled and removes some amount of potential bias. Additionally, since we later use the triangulation for distance metrics, a Delaunay triangulation provides a measure of distance closer to the actual Euclidean distance within the free space. The way that the triangulation is calculated is described in section 3.3.

Given a triangulation, we no longer need to search the whole space, instead, we sample the triangles using algorithm 4 which can be seen on page 38. It involves first computing the areas of the triangles, then picking a triangle at random weighted by their area. This is done to ensure uniform sampling of the area even though the triangles are of different sizes. Once a triangle has been selected, a random point is selected from within that triangle.

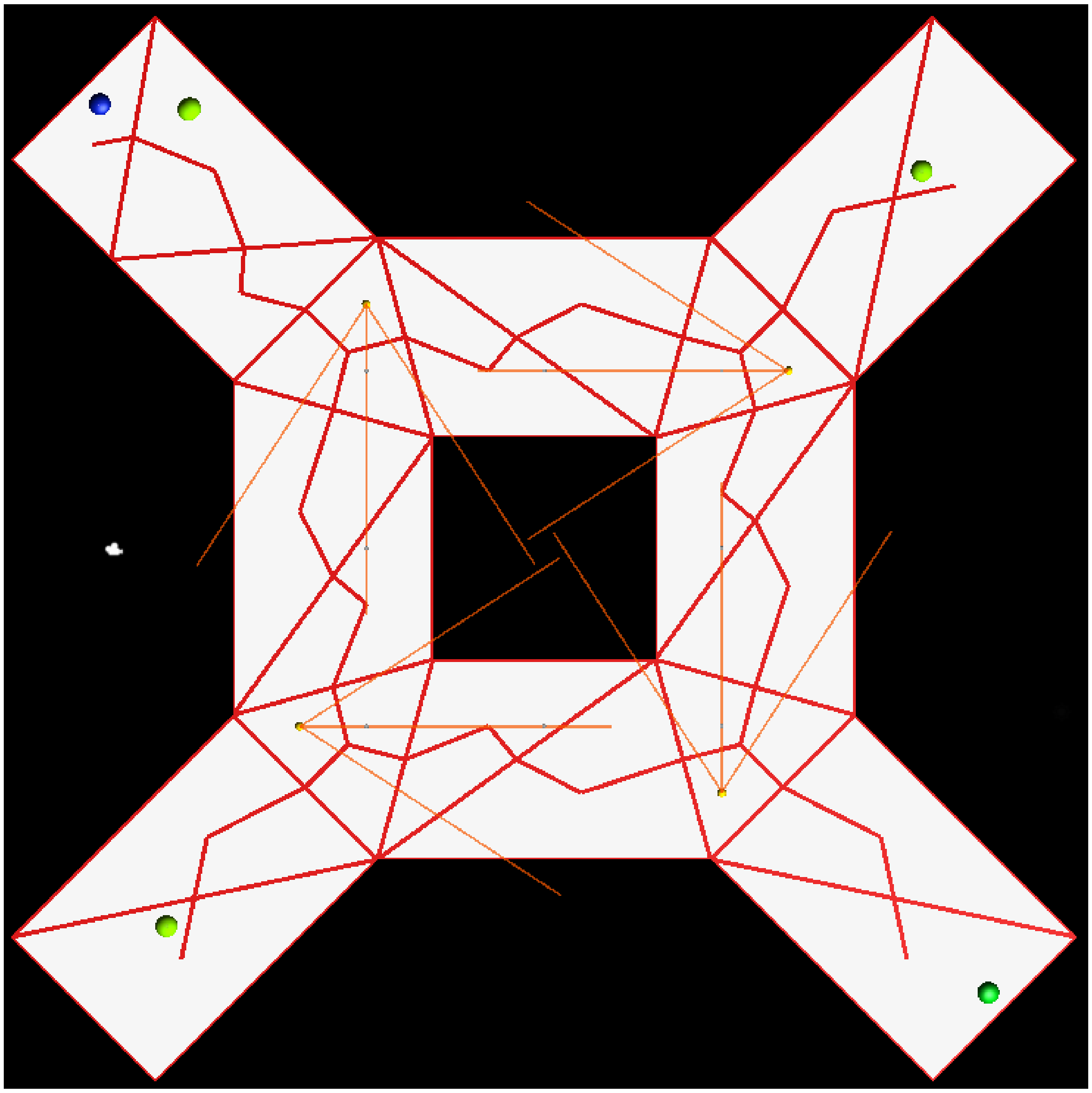


Figure 3.9 The Multi-Alarm Level with Triangulation and Distance Lines Drawn

However, now that we are choosing triangles, the initial distance constraint using a bounding box no longer works, and a new triangle based one is required. For this distance metric, we first transformed the list of triangles into a tree with the triangle containing the starting point as the root of the tree. Then, the distance between two adjacent triangles was defined as the distance from the center of one, to the middle of their shared edge and then to the center of the second one. An example of a level with the triangulation and lines representing the distance calculations drawn can be seen in figure 3.9. The white space can be seen to be divided into red triangles with the distance lines also drawn in red between them.

Then, to determine which triangles to search, we started at the root, and go down each branch, keeping track of the distance so far, until we reach a triangle that is too far away in every branch. Every time we add a node that is within a previously un-visited triangle we add it to the visited list, and sets its distance to zero. Then we redo the search for the reachable triangles. The distance metric remains the same one as previously for the time dimension.

3.4.3 Multiple vs Single Search

In running prototypes, it was found that at a certain point, adding more nodes to a single search became less helpful. Once the tree structure obtains a certain form, it becomes increasing less likely that the correct sequence of nodes will be found that will solve the level. This is hypothesized to be because when the tree has no or few nodes, the addition of each new node greatly increases the space covered by the tree, and thus the chance of finding a solution. However, if there is already a significant tree structure, additional nodes connect to the existing tree structure, filling in known areas more than extending the search into unknown territory. Thus, after a lot of nodes have been added, it becomes much more difficult to broadly change where the tree is going, and small changes are insufficient to lead to quick success. Due to this finding, it was found that a better way to get a reasonably good success rate and run time is to run more searches with fewer nodes as opposed to fewer searches with more nodes. In figure 3.10, we can see an example of an RRT search on an empty space. The broad structure is established after a few iterations, and successive iterations just fill in finer detail. In our context, if the broad structure is unfavourable, the detailed structure is unlikely to help find a solution.

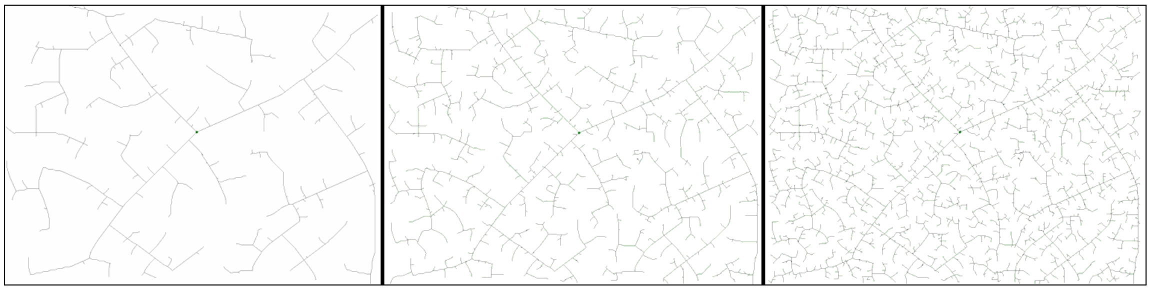


Figure 3.10 An RRT Search after a series of iterations [11]

3.4.4 Compositional Forms of the Search

Once more complicated levels were introduced, it was found that the search either failed completely, or just had a very low success rate. This is due to multiple reasons. Firstly, if there is only a small path through the level, and large search space, the longer the path, the harder it is to find. Additionally, as the level becomes bigger, there is more space that the nodes are sampled from, and thus the probability of sampling good nodes decreases. The search continues to sample in the areas that has found a path through because it does not know if the path it found is a good one.

In order to deal with more complicated levels a compositional search was created. The main idea is to break up the overall search into two smaller searches based on a heuristic half-way point in the solution path. If the level is larger than a given size, instead of doing a single search to get to the end, we find the ‘midpoint’ of the level, and do one search to reach it. It is not the true midpoint, but an approximation of a point near the middle of a path from the start to the end. This midpoint is expressed only in two dimensions, much like the start and the end, as it does not matter when we reach it, at least not if it is actually on a solution path. Once we have reached it, we do a second search to try to reach the end starting at the midpoint, with the state given by the result of the first search.

In order to find this midpoint, we used the triangulation’s distance metric to first find the distance to the triangle containing the end point from the starting triangle. Then, we look for the triangle whose distance is the closest to half of the distance to the end.

After creating this midpoint method, it was determined that it was possible that it could choose a bad midpoint for multiple reasons. For one thing, the method is based solely on distance from the start and does not factor in the distance to the end. This could theoretically lead to choosing a midpoint that is farther away from the end than the start is if they are not at opposite edges of the level. This can be seen in figure 3.11, where the point marked A would be chosen over the point marked B, because it is closer to half the distance from the start even though it is much further from the end. A better approach would be to base our midpoint choices on ones half-way along a geometric path from start to end. This is complicated by the fact that if there are multiple paths to end, but some of them are impassable due to guards, the midpoint may be chosen along such a bad path, and then the search from the midpoint to the end would be doomed to failure.

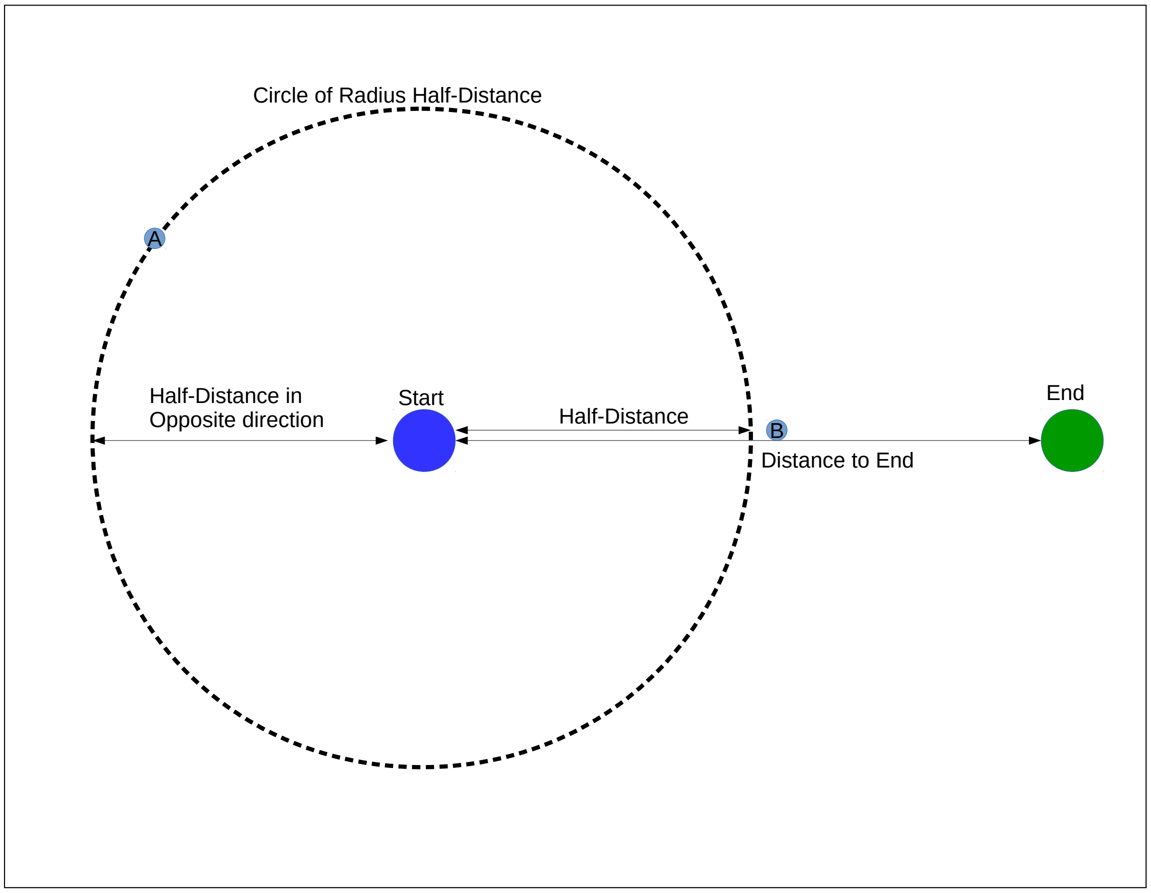


Figure 3.11 An Example of Bad Midpoints

Our approach is thus to consider multiple midpoints. We first find all simple paths from the start to the end based solely on the geometry of the level, ignoring guards since their behaviour can change. For details on finding these simple end paths, refer to section 3.3. Then, instead of simply computing one midpoint, we find one midpoint for each simple path from the start to the end. Different paths may share midpoints and so we also eliminate any duplicate midpoints to leave only the unique ones. With midpoints determined we can then perform the two part search for each midpoint in turn until we either fail with all of them or find a solution. We call this the multiple midpoint method. Note that while in this way we can guarantee at least one of our searches is aimed at a midpoint that is part of an actual solution (if one exists), an actual stealthy solution may be non-simple and require backtracking, or be otherwise unbalanced in complexity before and after the geometric midpoint, and so we have no guarantee our midpoint divides the search cost in two.

Algorithm 1 Triangulation : Composed of two procedures. The first finds the triangulation edges using the vertices and obstacles. The second part groups the edges into triangles

```
 1: procedure GetLines($O, V$)
 2:     $L \leftarrow new\ List < Line >$
 3:     for $i = 0$ to $V.size$ do
 4:         for $j = i + 1$ to $V.size$ do
 5:             $v1 \leftarrow V[i]$
 6:             $v2 \leftarrow V[j]$
 7:             if $\neg$v1$.Equals(v2)$ then
 8:                 $l \leftarrow Line(v1, v2)$
 9:                 $intersection \leftarrow False$
10:                 for all $o$ in $O$ do
11:                     if $o.intersects(l)$ then
12:                         $intersection \leftarrow True$
13:                         $break$
14:                 if $\neg intersection$ then
15:                     $L.Add(l)$
16: procedure FormTriangles($L$)
17:     $T \leftarrow new\ List < Triangle >$
18:     for all $l1$ in $L$ do
19:         $v1 \leftarrow l1[0]$
20:         $v2 \leftarrow l2[0]$
21:         for all $l2$ in $L$ do
22:             if $l1.Equals(l2)$ then
23:                 $continue$
24:             $found \leftarrow False$
25:             if $l2[0].Equals(v2)$ then
26:                 $found \leftarrow True$
27:                 $v3 \leftarrow l2[1]$
28:             else if $l2[1].Equals(v2)$ then
29:                 $found \leftarrow True$
30:                 $v3 \leftarrow l2[0]$
31:             if $found$ then
32:                 for all $l3$ in $L$ do
33:                     if $l1.Equals(l3)$ or $l2.Equals(l3)$ then
34:                         $continue$
35:                     if $(l3[0].Equals(v1)$ and $l3[1].Equals(v3))$ or
36:                             $(l3[0].Equals(v3)$ and $l3[1].Equals(v1))$ then
37:                         $t \leftarrow$ new $Triangle(v1, v2, v3)$
38:                         if $\neg T.Contains(t)$ then
39:                             $T.Add(t)$
```

Algorithm 2 Constructing the adjacency for the triangle.

```
procedure ConstructAdjacency(T)
    for all $t1$ in $T$ do
        for all $t2$ in $T$ do
            $AddAjacentst1, t2$
procedure AddAdjacents(t1, t2)
    if $t1.Equals(t2)$ then
        $continue$
    for all $l1$ in $t1.lines$ do
        for all $l2$ in $t2.lines$ do
            if $l1.Equals(l2)$ then
                $t1.AddNeighbour(t2)$
procedure AddAdjacentsBothWays(t1,t2)
    AddAdjacents(t1,t2)
    AddAdjacents(t2,t1)
```

Algorithm 3 Converting the triangulation to a Delaunay one with updated adjacencies.

```
procedure DELAUNAYIFY(T)
    $done \leftarrow False$
    $edgeFlipped \leftarrow False$
    while $\neg done$ do
        for all $t1$ in $T$ do
            for all $t2$ in $t1.neighbours$ do
                $lc \leftarrow sharedEdge(t1,t2)$
                $l11 \leftarrow t1.edgeWith(lc[0])$
                $l12 \leftarrow t1.edgeWith(lc[1])$
                $l21 \leftarrow t2.edgeWith(lc[0])$
                $l22 \leftarrow t2.edgeWith(lc[1])$
                $v11 \leftarrow l11[1] - l11[0]$
                $v12 \leftarrow l12[0] - l12[1]$
                $v21 \leftarrow l21[0] - l21[1]$
                $v22 \leftarrow l22[0] - l22[1]$
                $angle1 \leftarrow Angle(v11,v12)$
                $angle2 \leftarrow Angle(v21,v22)$
                if $(angle1 + angle2) > 180f$ then
                    $T.remove(t1)$
                    $T.remove(t2)$
                    $t3 \leftarrow new\ \ Triangle(l11[0],l11[1],l12[0])$
                    $t4 \leftarrow new\ \ Triangle(l21[0],l21[1],l22[0])$
                    for all $tt$ in $t1.neighbours$ do
                        $tt.neighbours.Remove(t)$
                        $AddAdjacentsBothWays(tt,t3)$
                        $AddAdjacentsBothWays(tt,t4)$
                    for all $tt$ in $t2.neighbours$ do
                        $tt.neighbours.Remove(t)$
                        $AddAdjacentsBothWays(tt,t3)$
                        $AddAdjacentsBothWays(tt,t4)$
                    $T.Add(t3)$
                    $T.Add(t4)$
                    $edgeFlipped \leftarrow True$
            if edgeFlipped then
                break
        if edgeFlipped then
            $edgeFlipped \leftarrow False$
        else
            $done \leftarrow True$
```

Algorithm 4 An algorithm for random sampling based on a triangulation. We first compute relative triangle areas in SetUpSampling, and then each time we sample we call SampleTris to get an actual point.

```
procedure SETUPSAMPLING(T, minT, maxT)
    $standardizedAreas \leftarrow new\ List < float >$
    $areas \leftarrow new\ List < float >$
    $areaSum \leftarrow 0$
    for all $t1$ in $T$ do
        $lins \leftarrow t1.getLines$
        $l1 \leftarrow lins[0].Magnitude$
        $l2 \leftarrow lins[1].Magnitude$
        $l3 \leftarrow lins[2].Magnitude$
        $s \leftarrow 0.5f \times (l1 + l2 + l3)$
        $area \leftarrow Sqrt(s \times (s - l1) \times (s - l2) \times (s - l3))$
        $areas.Add(area)$
        $areaSum \leftarrow areaSum + area$
    $sumSoFar \leftarrow 0$
    for all $a$ in $areas$ do
        $standardizedAreas.Add((a + sumSoFar)/areaSum)$
        $sumSoFar = sumSoFar + a$
procedure SAMPLETRIS(T)
    $SetUpSampling(T)$
    $trisInd \leftarrow Random.Range(0, 1)$
    for $i = 0$ to $standardizedAreas.Count$ do
        if $trisInd \leq standardizedAreas[i]$ then
            $break$
    $tri \leftarrow T[k]$
    $rl1 \leftarrow 1$
    $rl2 \leftarrow 2$
    while $rl2 \geq rl1$ do
        $rl1 \leftarrow Random.Range(0, 1)$
        $rl2 \leftarrow Random.Range(0, 1)$
    $v0 \leftarrow tri.vertex[0]$
    $v1 \leftarrow tri.vertex[1] - tri.vertex[0]$
    $v2 \leftarrow tri.vertex[2] - tri.vertex[1]$
    $point \leftarrow v1 + rl1 \times v2 + rl2 \times v3$
    $SampleX \leftarrow point.x$
    $SampleY \leftarrow point.y$
    $SampleT \leftarrow Random.Range(minT, maxT)$
```

Chapter 4
Experiments

In this chapter we describe the experiments that were run to test our implementations and motivate new changes. To do this, we first describe each of the levels that we created or modeled to run experiments on. We then proceed with an explanation of what each of the experiments were and a discussion of the results of each experiment.

4.1 Level Descriptions

In this section we describe each of the levels that were created for experiments to be run, as well as the motivation for their creation. These levels are intended to include situations representative of, but also generic to many stealth games. All of these levels are intended to be unsolvable without the use of a distraction.

Our suite includes six levels: Simple Level, State of Decay Level, Thief Level, Multi-Alarm Level, Two-Part Level and Two-Part Choice Level. This array of levels includes simple test levels, more complicated ones that require solving multiple guard-distraction interactions, and two levels modeled from existing games. Note that we make use of more synthetic levels than levels based on existing games. In general, proprietary games do not tend to release the exact schematics of their games and stealth levels. Our models of actual game levels are thus based on video playthroughs created by fans. Fan-created videos, however, tend to focus on optimal or visually interesting playthroughs, with the actual stealth situations located arbitrarily or incidentally, making good extraction a labour-intensive process. This also affects reliability. In not focusing on the full and exact details of a stealth problem, precise and formal reconstructions of complex game levels from fan videos is difficult, often leaving ample room for interpretation, which can easily obviate the authenticity of the source. Use of synthetic levels allows us to illustrate a broader range of stealth problems, especially for investigating more complex scenarios. Useful future work in stealth analysis could include a comprehensive extraction of stealth problems from actual games.

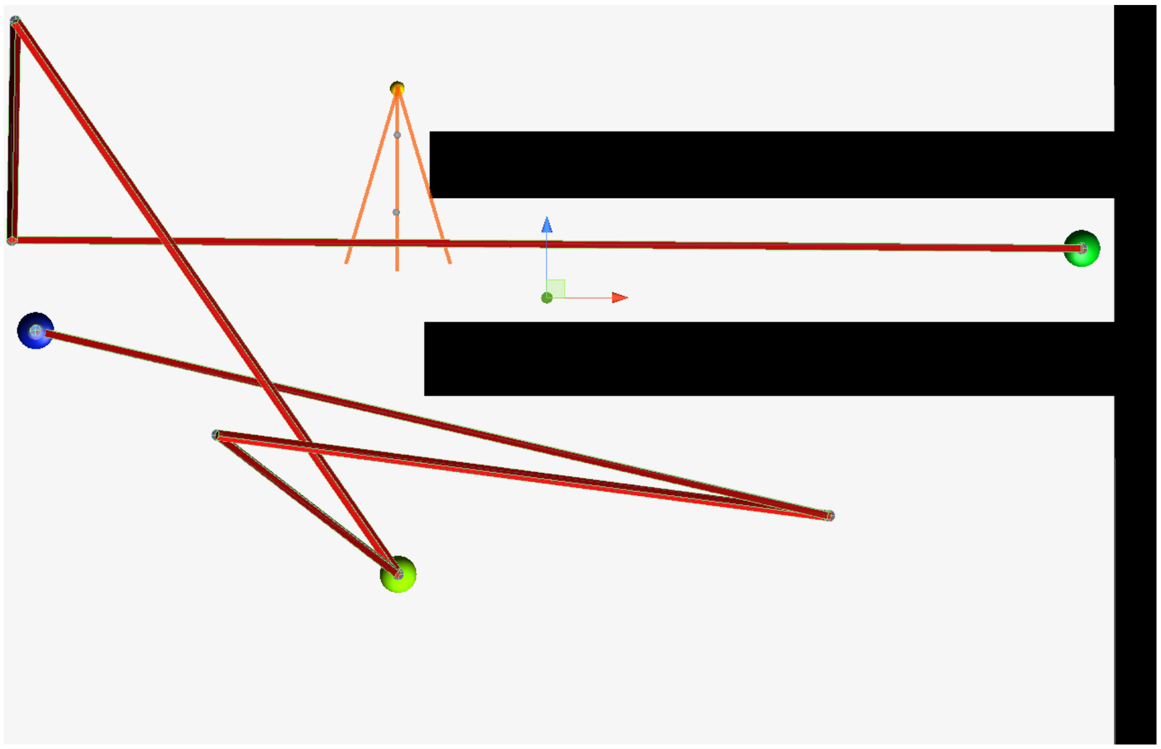
4.1.1 Simple Level

In figure 4.1, we see a simple proof of concept level. The player starts at the blue circle on the left. The free space is represented by the colour white, while the obstacles are represented by black shapes. There is a corridor down which is the end of the level, represented by a dark green circle. A guard patrols in front of it, such that it is not possible for the player to enter the corridor without encountering their field of view. The guard is represented by a small yellow circle, with orange lines representing the edges of the field of view (and moving up/down). There is a distraction, that when the player activates it, the guard goes to that same point before returning to patrol. This is an example of variation 1 from table 3.1. The distraction is represented by the light green circle. This could represent the player making a noise, perhaps by stepping on broken glass, in the distraction location, thus attracting the guard. The red line represents a possible solution, flattened in time due to the overhead view.

4.1.2 State of Decay Level

This level, seen in figure 4.2 is modeled on a portion of a real game level from *State of Decay* as seen in a Youtube clip [10]. There is a starting area on the left that is fenced off from stationary enemies near a house. The player throws a distraction from a point in the fenced off area, all the enemies go to it, and then he sneaks into the house. It was thus modeled with the location to throw the distraction from as the distraction point, and a good place to throw it being the resulting location that the enemies go to when it is activated.

Figure 4.1 Simple Level



Both the distraction activation point and the distraction event point are represented with the light green circles. The actual level is an instance of variation 8 from table 3.1, however we implement it as an instance of variation 5. The difference is, in the actual level there is a delay between when the player throws the distraction, and when it lands. However, this time difference is just the length of the throwing animation, and is small enough to be insignificant.

Although quite simple, the use of distraction here is common to multiple stealth games, and we were able to find another game that has a highly similar level to this one. This was in the game *The Evil Within*, and can be seen in another Youtube clip [9]. The difference seen between them is that in *The Evil Within* the enemies are slowly moving before the player distracts them rather than stationary, but the mechanics are the same.

Figure 4.2 State of Decay Level

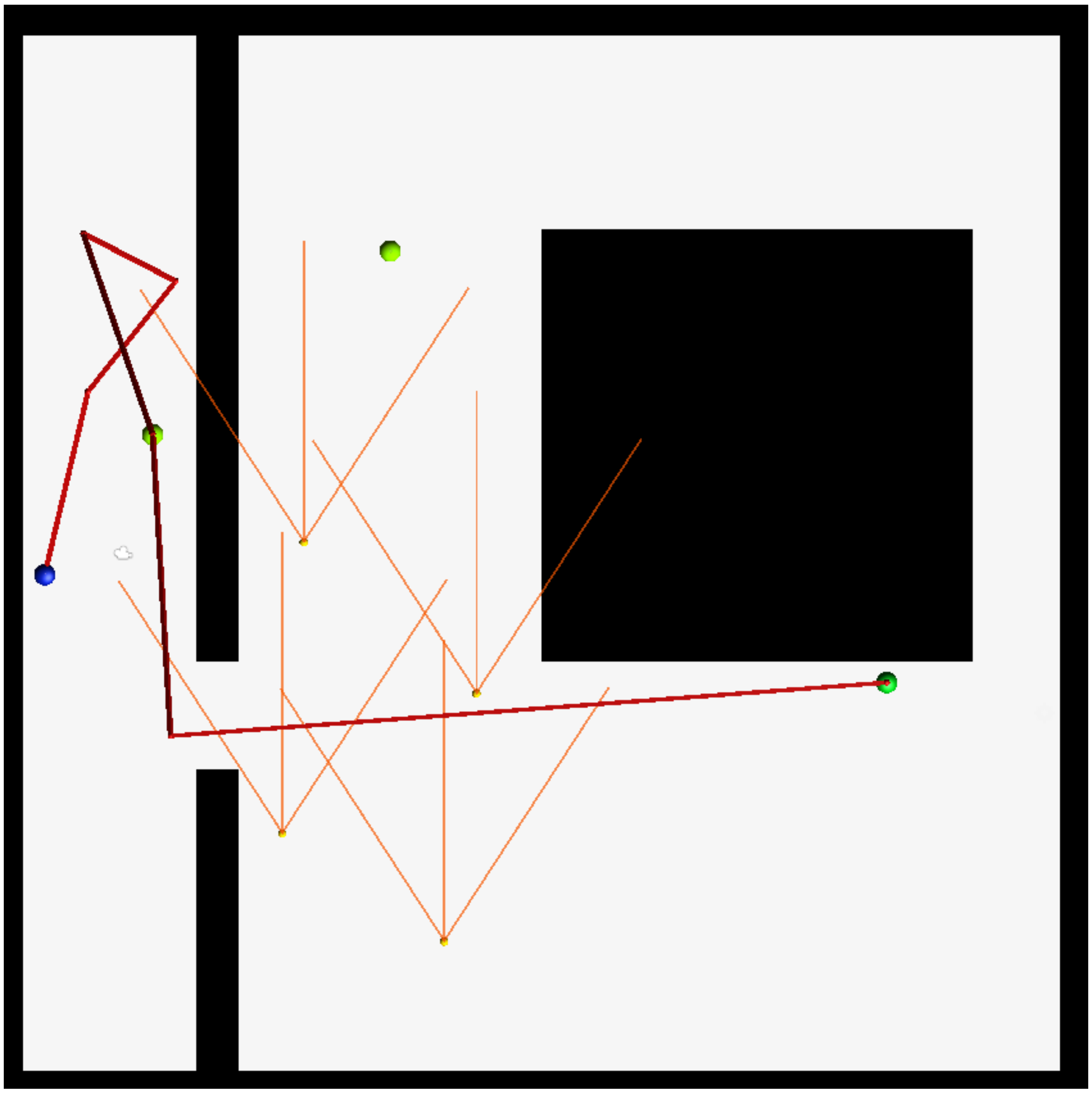
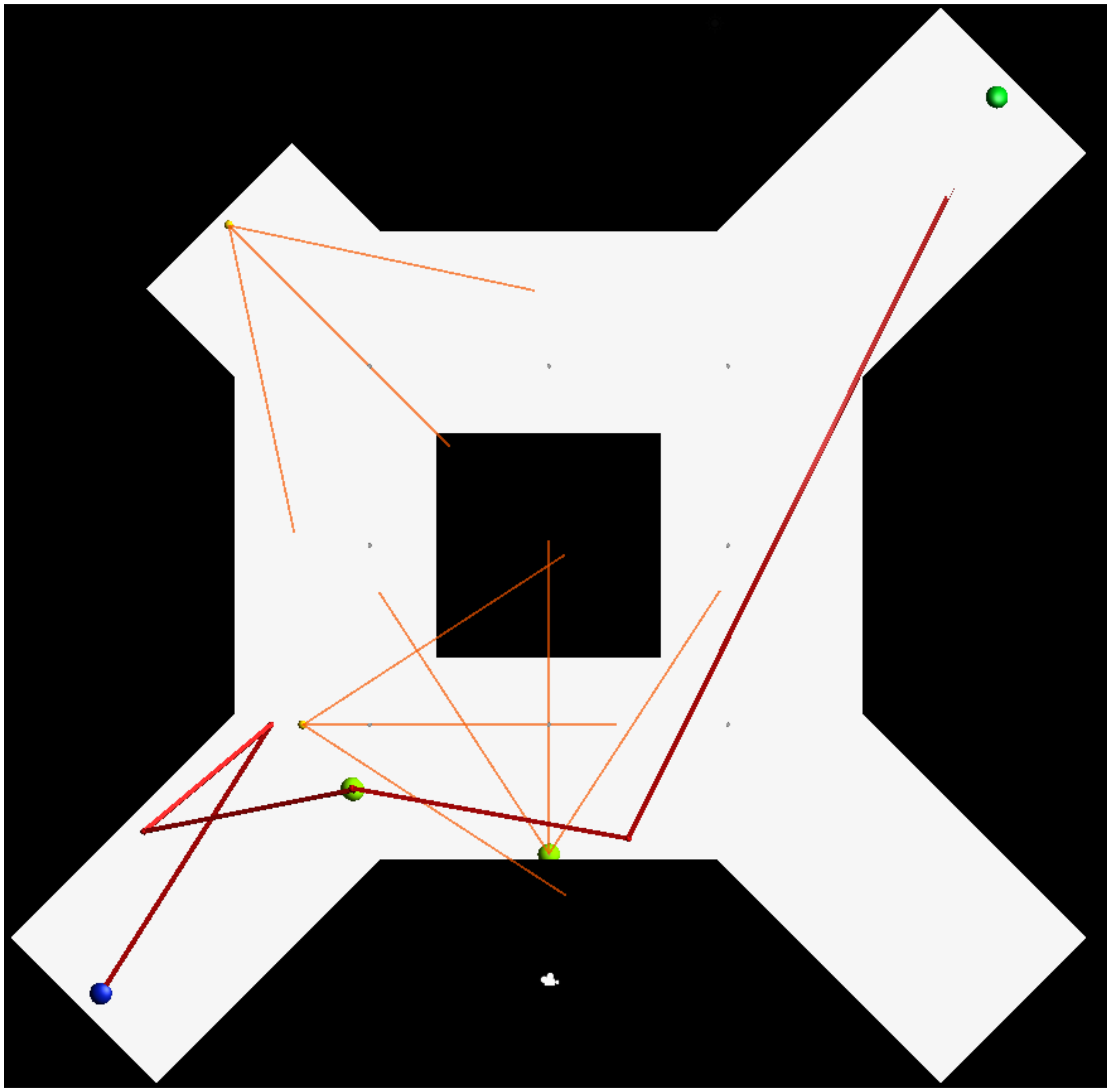


Figure 4.3 Thief Level



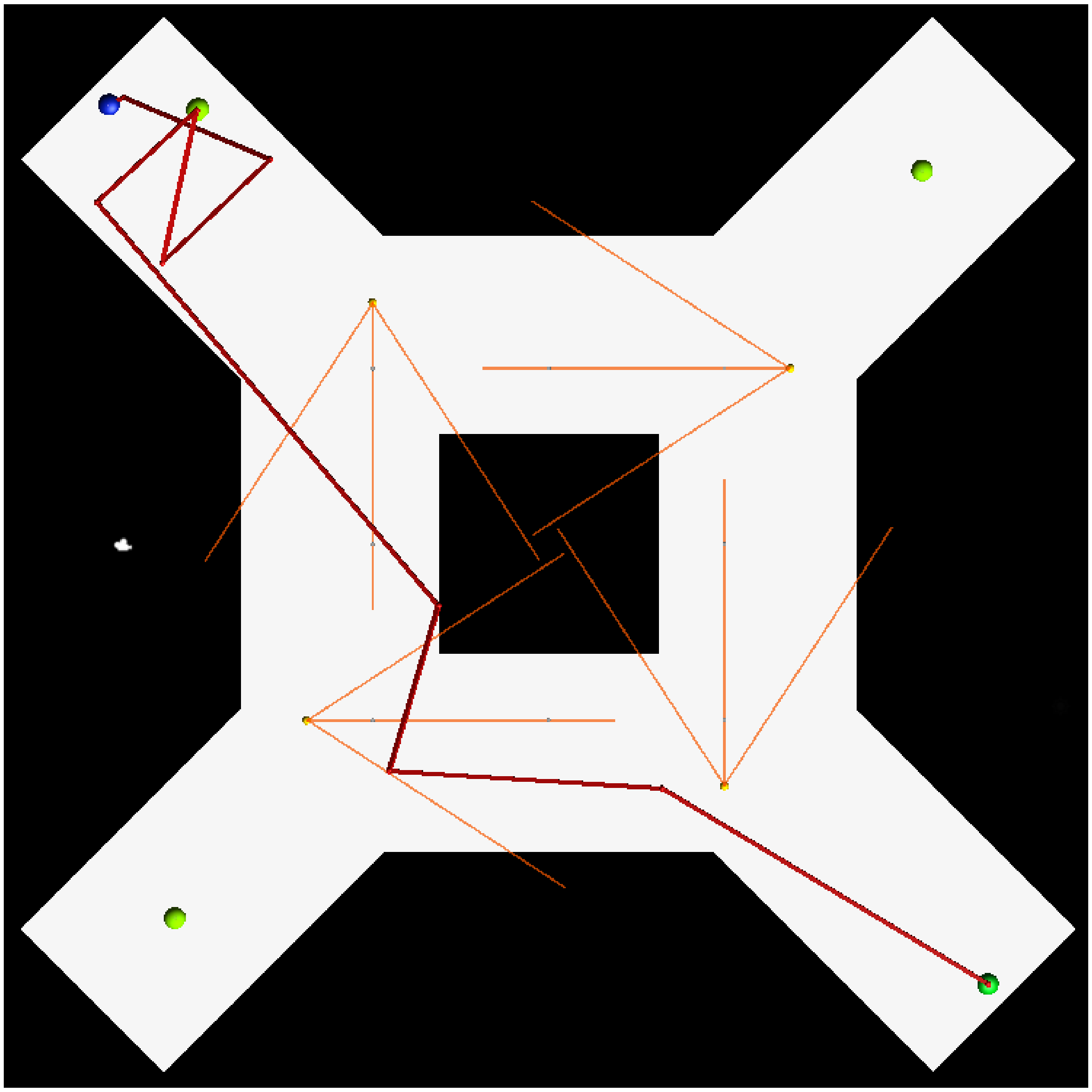
4.1.3 Thief Level

This level, seen in figure 4.3 is modeled on a portion of a real game level from *Thief* as seen in a Youtube clip [4]. The starting area is in the bottom left corner. There is a stationary guard in the top left, and another stationary guard in the middle of the bottom. Finally, the third guard starts in the bottom left, and patrols in a square around the middle. The player can cause a distraction that makes the bottom guard turn around which allows the player to sneak through. In the video, the player shoots a “water arrow,” putting out a fire behind the guard and causing him to turn around to face the wall. We model this abstractly as a distraction. Both the distraction activation point and the distraction event point are represented with the light green circles. Much like the State of Decay Level, this is an instance of variation 8 from table 3.1 which we model as an instance of variation 5 by ignoring the animation time. In this case, that is the animation of the arrow being shot by the player into a fire behind the guard.

4.1.4 Multi-Alarm Level

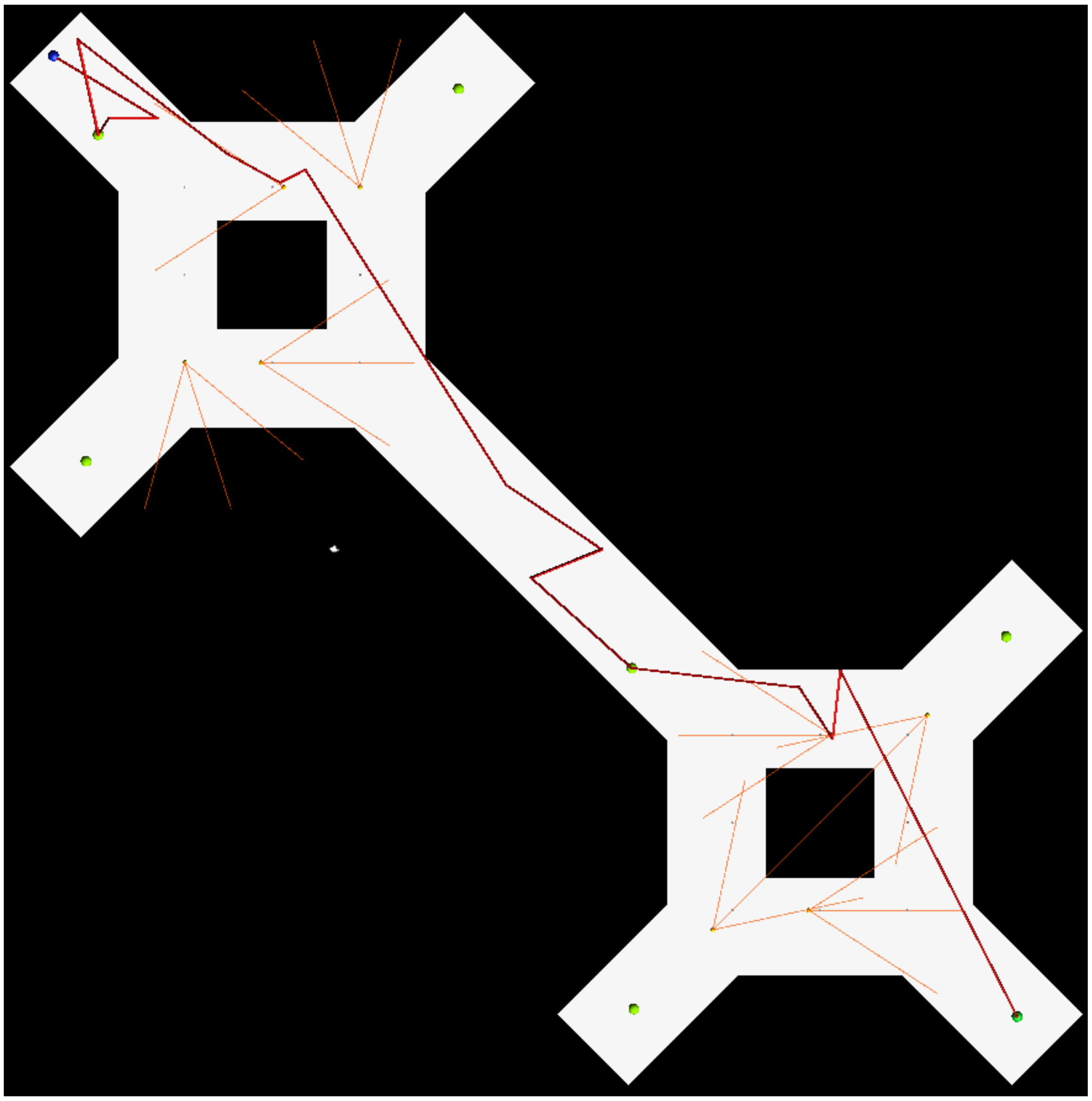
This is a more complicated test level created to show more versatility in terms of both what the search can accomplish, and what kinds of levels and distractions can be modeled. It is more complicated due to having more guards, and more distraction event locations as well as by simply being larger. It is displayed in figure 4.4. In this level there are four guards patrolling in a square around a central obstacle. There is a corridor branching from each of the four corners of the central region. Two opposite corridors lead to the start and the end, and the other two corridors lead to alarm points. The distraction activation point is near the starting location of the player. When it is activated, any guard for whom the closest corridor is one containing an alarm point will go to that alarm point before returning to patrolling around the central obstacle. In this way, this is showing that the location of the distraction activation does not necessarily match the location guards go to when distracted. Additionally, it means one distraction can lead different guards to different places. It is an example of variation 2 from table 3.1. It could also be argued that it could be an example of variation 6, because although the guards that are distracted go back to the same patrol route, they are now shifted in time, and so the total pattern of the guard patrols

Figure 4.4 Multi-Alarm Level



has changed. Due to this, some of the guards are now closer together, and some are farther apart instead of them being evenly distributed. A potential real scenario it could represent is the player setting off an alarm system, and the nearest guards needing to go turn it off before continuing to patrol.

Figure 4.5 Two-Part Level



4.1.5 Two-Part Level

The two-part level, seen in figure 4.5, is simply the two copies of the multi-alarm level attached end to end. It showcases a more complicated level where a compositional approach can be taken, in addition to actually having two separate, independent distractions that are both needed in order to solve the level. This has a complexity that is much more indicative of a full game, composed of a larger level with multiple stealth puzzles and sets of guards. In much the same way as the multi-alarm level, it is a more complicated version of variation 2 from table 3.1.

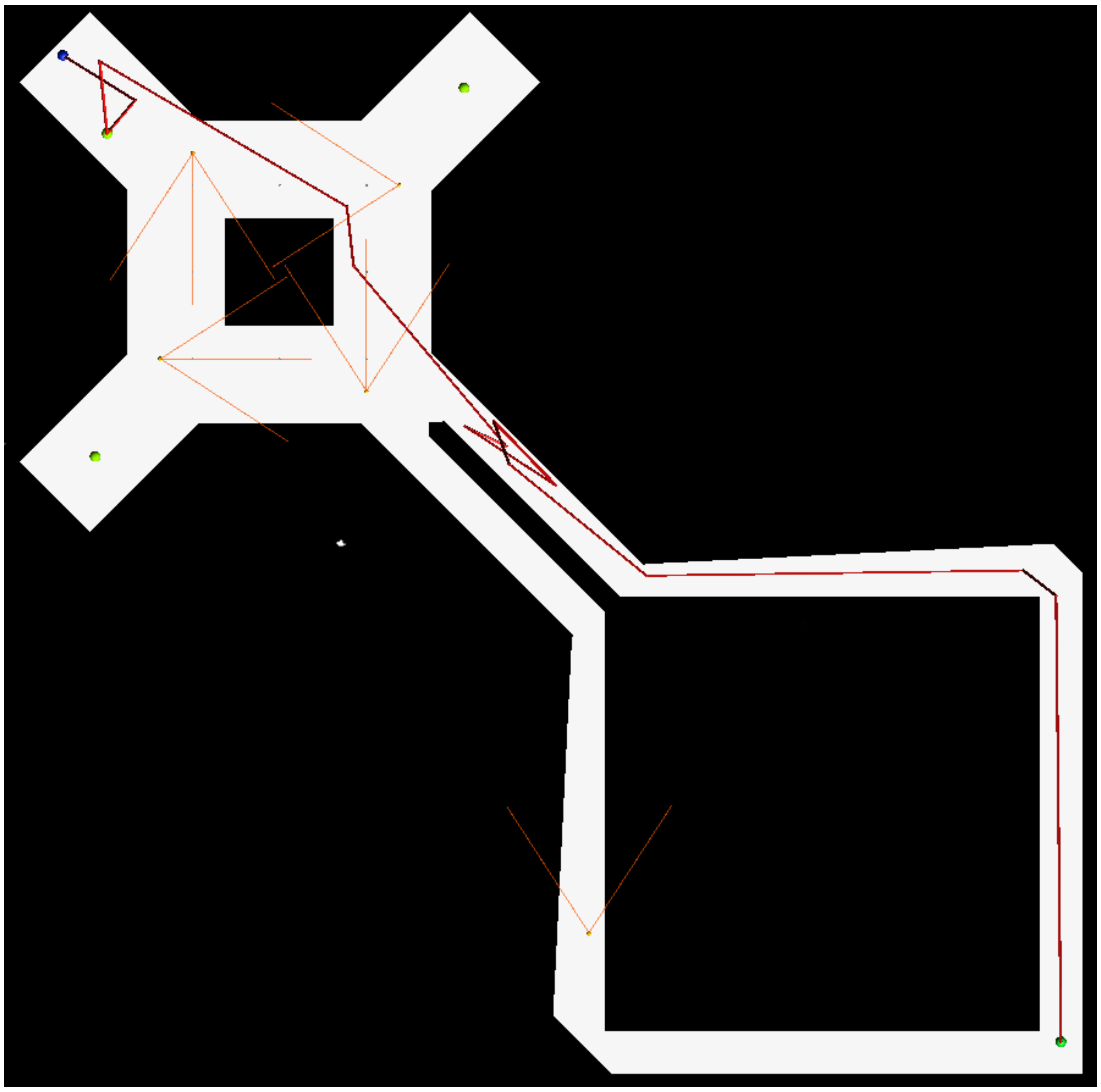
4.1.6 Two-Part Choice Level

The two-part choice level, figure 4.6, was created in order to determine how well our single-midpoint search fared when bad midpoints were present. The first half of the level is simply a multi-alarm level, but attached to the end there is a choice of two corridors to go down. One of them leads to the end of the level, while the other one seems to lead to the end of the level, but has a stationary guard blocking it who cannot be distracted. This is done to illustrate that although geometrically there are two equally viable midpoints, in this level only one leads to the solution. This level would be difficult for a player to solve in a stealth game, as the choice of which corridor to go down is not necessarily obvious ahead of time, and so multiple play attempts would likely be required. Much like the regular two-part level, the distraction in this level is an example of variation 2 from table 3.1.

4.2 Results and Discussion

In this section we describe the experiments that were run, and discuss their results. It is split into a series of subsections. The first describes the experiments that were run once to test the search. These include a search of a simple level, a search of a real game level, and motivation for the triangulation. The next section outlines the experiment that was run to determine whether more searches or more nodes within a single search give better results. Then, there is a subsection that investigates the effects of the improvements to the

Figure 4.6 Two-Part Choice Level



search that were described in section 3.4.2. Finally, the last subsection shows experiments that were run to show the effectiveness of the compositional search as described in section 3.4.4. In general our goal was to be able to solve levels with a high, >85% success rate and reasonable low times. These experiments were all run on a Windows 10 machine, using a Intel i7-6700HQ processor operating at 2.60GHz, with 16 GB of RAM using Unity3D 5.3.1.

4.2.1 Simple Experiments

Our initial experiments focused on ensuring we could successfully solve our more basic game levels. In these cases we did not deeply explore parametrization, using these results instead to show feasibility, and motivate further experimentation.

For a first, proof of concept experiment, we used the Simple Level with a single guard that needs to be distracted to reach the goal. 100 trials were performed with the full search with parameters of 2500 nodes, and one search attempt, and had a high success rate, at 97% in less than one second. As this was a high success rate and a very fast time, the parameters were not increased.

An important goal of our work is that we can solve levels found in actual games. Thus another experiment was to model a level from a real game, and try to solve it using our search method. For this we used the State of Decay Level. Our search solved it 99% of the time in less than one second. 100 trials were performed with the parameters of 4000 nodes, and five search attempts. Here, we gave it additional resources because at 2500 nodes and one search attempt the success rate was only 65%, and at 4000 nodes and one search attempt the success rate was only 70%.

We also had a second test level from a real game that we modeled, the Thief Level. We chose to run it with 2500 nodes and one search attempt as well as 4000 nodes and 5 search attempts like the State of Decay Level. With the 2500 nodes and one search we had a success rate of 88% in 0.1 seconds for success and with 4000 nodes and 5 search attempts, we had a success rate of 100% with an average time of 0.15 seconds after 100 trials.

As game levels often contain many obstacles, a search on the entire space would be likely to sample from within obstacles a significant portion of the time. Even if these samples are easily rejected as infeasible, they constitute wasted resources, subtracting from our node budget and increasing search time. Of course the effect depends very much on the level design, and may not be a concern in lightly occluded levels. Indeed, examining the behaviour of our experiments on the Simple Level, we found only 4% of nodes were within obstacles. The effect in the Multi-Alarm Level, however, was much more severe. In this case we observed 56% of the nodes sampled were within obstacles and thus wasted, in a search of 4000 nodes. Performance concerns with this larger level, supported by this brief analysis motivated our triangulation improvement.

Interestingly, even at this early stage in experimentation our analysis demonstrated useful results. While performing the searches, it was found that the Multi-Alarm Level is actually solvable *without* using distractions, albeit very rarely. An example of such a solution can be seen in figure 4.7. Guard fields of view do not fully overlap, and this allows a player to closely follow a guard around the bottom left corner, dashing to the goal before the next guard behind them turns that same corner. This was of course not our intention, and so illustrates the importance of using tools to verify game design. From a human, design perspective the level appeared to be unsolvable without the use of distractions, and this error only became apparent through the use of algorithmic, repeated and randomized search. However, this occurs very rarely and was only discovered when the search was run with distractions being disallowed over 100 trials and one such solution was found.

4.2.2 Effects of Resource Use on Searches

Understanding the impact of resource consumption in solving distraction levels is of course important. A given node (or time) budget can be applied in terms of performing many small searches, or it can be used to perform a single large search. Here we thus experiment with this trade-off.

For the high success rate for the State of Decay Level, we used 4000 nodes and five attempts. To show the effects of doing this instead of simply using a single search of 20 000 nodes we ran an experiment. In this experiment we tested the effects of simply increasing the maximum number of nodes to sample for the search versus increasing the number of searches. For each test, we ran 200 trials, and the baseline of the experiment was

Figure 4.7 Multi-Alarm Level Solution without Distraction

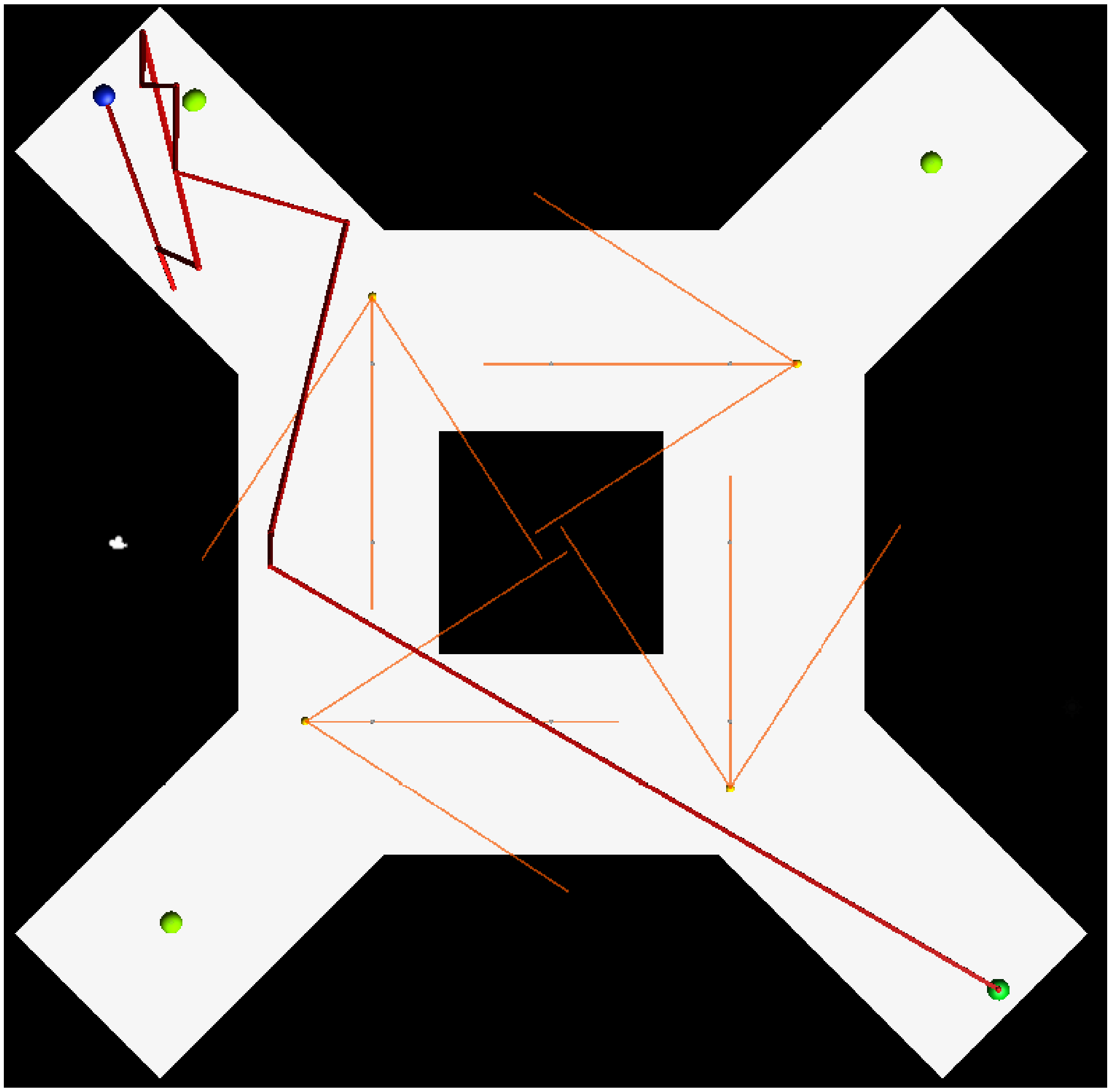


Table 4.1 Node and Search Number Variation on Multi-Alarm Level

Results of Variation in Resources						
Nodes Used	Searches Used	Success Rate	Success Time (s)	StDev	Failure Time (s)	StDev
4000	1	0.34	0.75	0.36	1.54	0.18
8000	1	0.34	0.90	0.44	2.323	0.24
16000	1	0.41	1.27	0.84	3.921	0.25
32000	1	0.56	1.77	1.84	7.353	0.36
4000	2	0.54	1.56	0.76	3.450	0.24
4000	4	0.82	3.10	1.63	6.969	0.46
4000	8	0.97	3.34	2.80	12.99	0.34

Figure 4.8 Increasing Resources and Success Rate

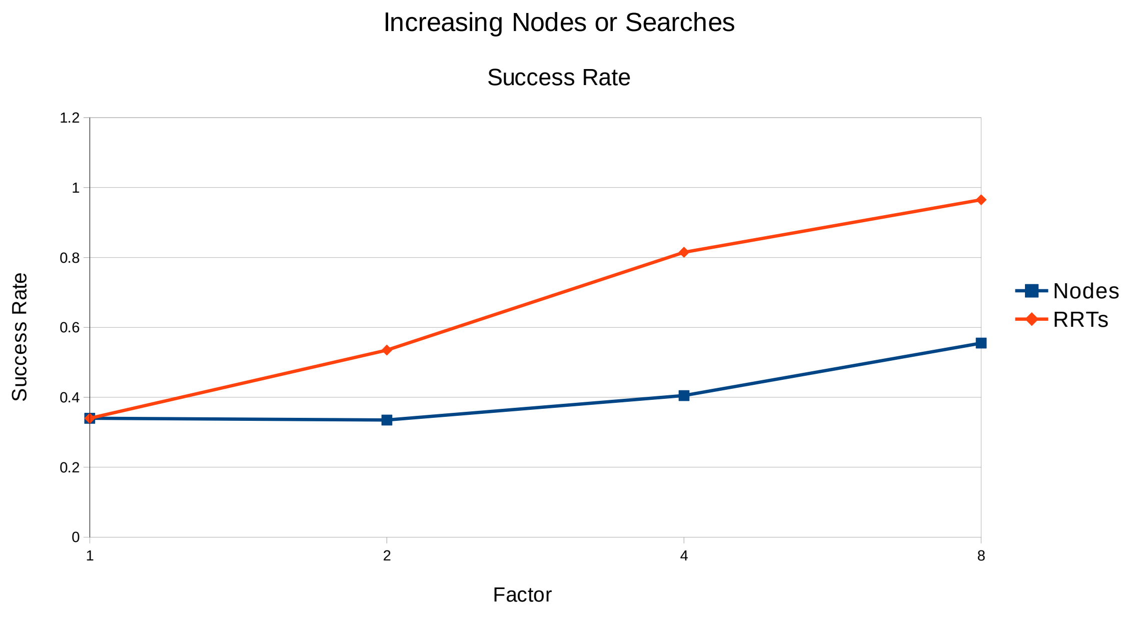


Figure 4.9 Increasing Resources and Average Success Time

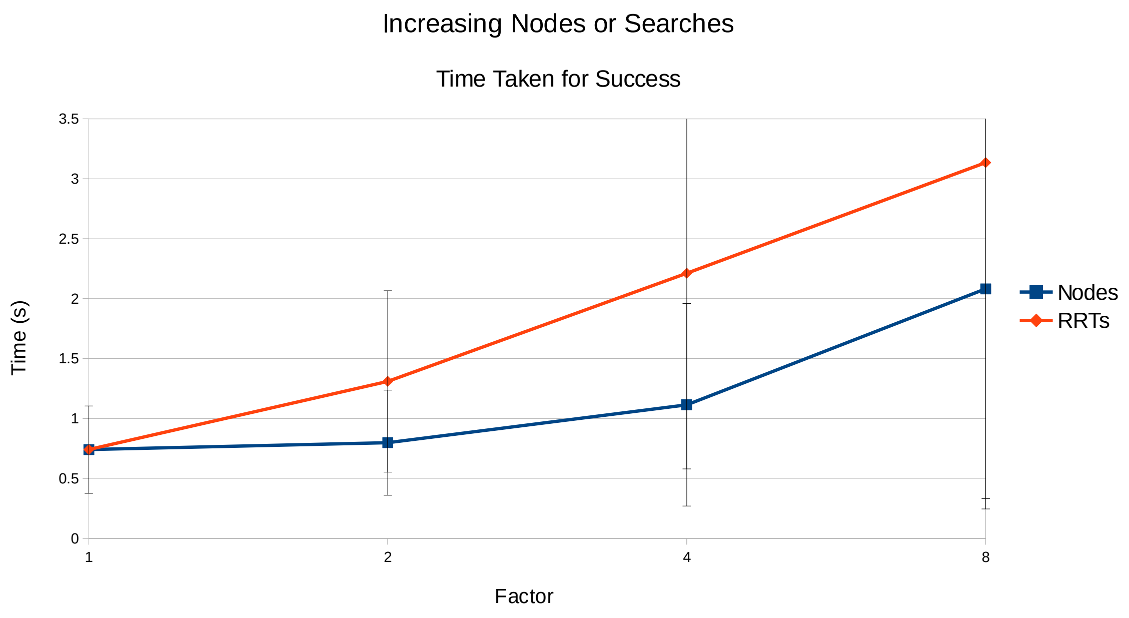
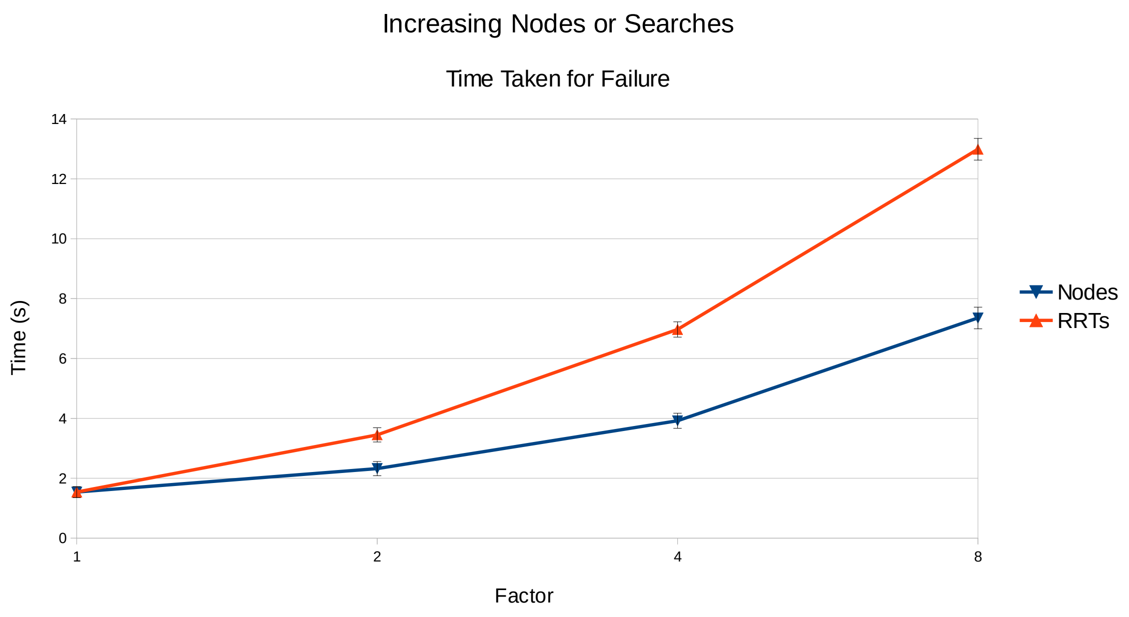


Figure 4.10 Increasing Resources and Average Failure Time



4000 nodes and one search, and then either the number of nodes, or the number searches was increased to 2-fold, 4-fold, and 8-fold. The test was run on the Multi-Alarm Level because it was seen to be sufficiently difficult to merit the use of more nodes and/or more searches. The Simple Level was unsuitable as it already had a 97% success rate with only 2500 nodes. The results can be seen in table 4.1.

The trends for success rate can be seen in figure 4.8, with error bars representing standard deviation. It was found that increasing the number of nodes does increase the success rate, but not by very much, increasing to 58% from 37% when increasing the number of nodes used was increased by eight times. However, when instead eight searches were used with the base number of nodes, the success rate increased to 95%. Thus, it can be seen that doing multiple searches is a better way to increase the success rate. This supports our hypothesis, outlined in subsection 3.4.3, that our solutions are based on the coarse structure of the tree which is established early on, and adding further fine detail becomes less likely to lead to a solution as the construction of the tree goes on. It may even be true that using even less nodes and more attempts would be a more efficient use of resources, although the threshold likely strongly depends on the specific problem.

Another trade-off exists in terms of analysis time. Even with the same node budget, there is overhead in doing each search, which may or may not be worse than the cost of doing neighbourhood queries in an increasingly large, single tree. We therefore measured the time taken, with separate averages computed for time taken to succeed and time taken to fail. The success time graph can be seen in figure 4.9, with the error bars once again representing standard deviation. The success time was greater for the multiple searches than for the increased number of nodes in the search. However, the standard deviation was very high, and although the trend in averages shows increasing number of searches to be more expensive than increasing the tree size, it is hard to claim this difference is significant. A high variance in successful searches is not surprising of course. It is most likely due to the highly variable nature of RRT in general, as it is based on being random. Note that in addition to success time, the number of nodes sampled was also calculated for the successful trials, and it too had an extremely high standard deviation for much the same reasons as the success time.

The time taken to fail, which can be seen in figure 4.10, however, more accurately represents the increase time cost of using more nodes, or more searches. Here, it is seen that the time increases at a significantly greater rate when increasing the number of searches. This is likely due to the overhead of starting a new search, as well as the fact that while the data structure for nearest neighbour queries has good performance asymptotically, it is necessarily less efficient at smaller sizes. At the 8-fold increase, the multiple RRT method takes almost double the time as just increasing nodes. However, the multiple RRT at 4-fold increase has similar times to the increased node method at the 8-fold level while still having a much higher success rate.

An important note lies in our definition of failure and success. We simply use a hard cut-off in terms of resources after which if we have found a solution we succeed, and if we have not we fail. In theory, we could be quite close to success when we declare failure, or could have reached a state where success is impossible (or at least impossible within our resource constraints) long before we run out of resources and declare failure. However, without having some sort of extra knowledge it is not possible to know how close or far we are from a solution, and by determining that multiple smaller searches have a higher success rate, we are attempting to optimize our use of resources in our search for success. Furthermore, since we perform multiple searches, the combination of the success rate and the resource limit provide an indication of the difficulty of the level to our search. Additionally, if this was ever to be used within an actual game, or in any live context, it is quite reasonable to have hard limits on the resources available for the search at which time success or failure must be declared.

4.2.3 Effects of Improvements to the Search

Our initial experiments with our basic search on the Multi-Alarm Level showed poor results. The basic search (with 4000 nodes and 5 searches) only had success rate of 16%, and took about 10 seconds to find a solution. We desired both a higher success rate and a faster time, and so developed and applied different improvements, as described in the previous chapter. In this subsection we show the effects of the improvements we made to the search. We first describe the details of how the experiment was run, followed by a discussion of the effects on success rate. Then, we discuss the effects of the improvements on other metrics such as time taken and resources used. We then conclude with a summary of the effects of these different improvements.

Table 4.2 Different Search Results for Multi-Alarm Level

Results of Different Searches					
Name	Success	Searches	Nodes	Success Time(s)	Failure Time(s)
BasicSearch	0.16	2.88	8362	9.97	19.95
Standard Deviation		1.54	6452	6.62	7.65
BasicDSearch	0.53	2.70	7572	30.02	54.16
Standard Deviation		1.45	5875	19.88	10.92
DDSearch	0.12	2.75	8755	2.75	5.01
Standard Deviation		1.54	6053	1.57	0.48
SimpTriSearch	0.70	2.49	6565	24.99	54.50
Standard Deviation		1.29	5204	15.63	13.06
FullSearch	0.87	2.17	5417	2.62	7.67
Standard Deviation		1.31	5365	2.08	0.40

The Experiment

Due to its poor basic performance we used the Multi-Alarm Level in order to determine the effects of various improvements to the RRT search. For this, the comparison was made between the basic search (BasicSearch), basic search with prevention of distraction stacks (BasicDSearch), basic search with distance metric and distraction stack prevention (DDSearch), triangulated search with distraction stack prevention (SimpTriSearch), and finally, triangulation with distraction stack prevention and distance metric (FullSearch). For each of the experiments 100 trials were done, and the same parameters were used which were 4000 nodes and five search attempts.

The results of the search can be seen in table 4.2. For each experiment we have measured success rate, the average number of searches used for a successful trial, the total number of nodes used on average in a successful trial, the average time for a successful trial and the average time for an unsuccessful trial. For the nodes, we sum the number of nodes used over all searches within a trial, and then take the average over all trials.

Figure 4.11 Success Rate of Different Searches

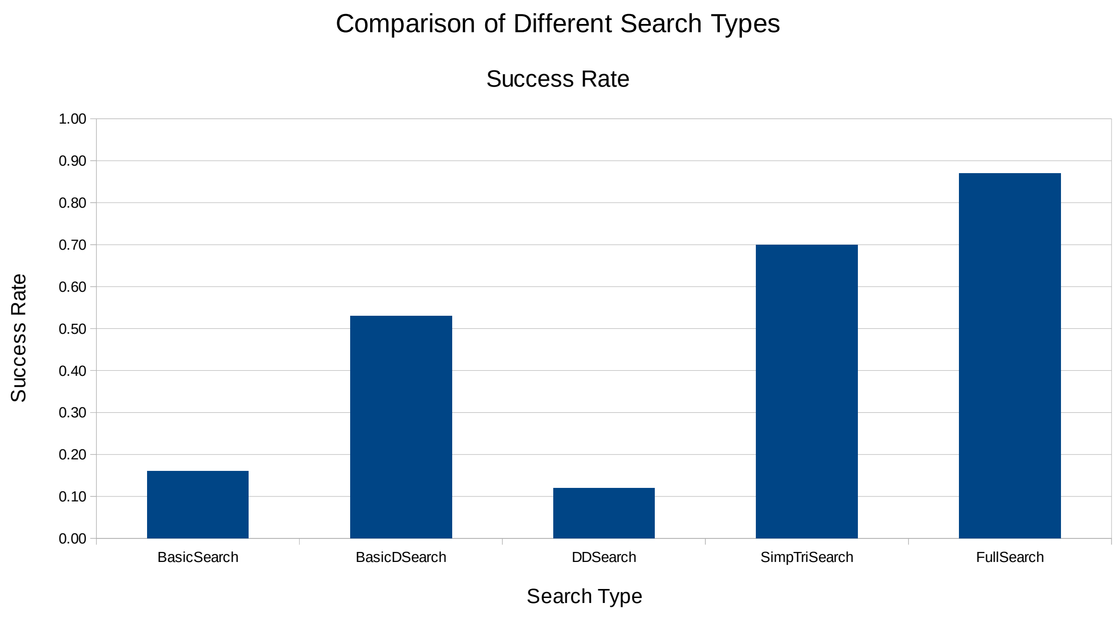


Figure 4.12 Average Nodes Used in Different Searches

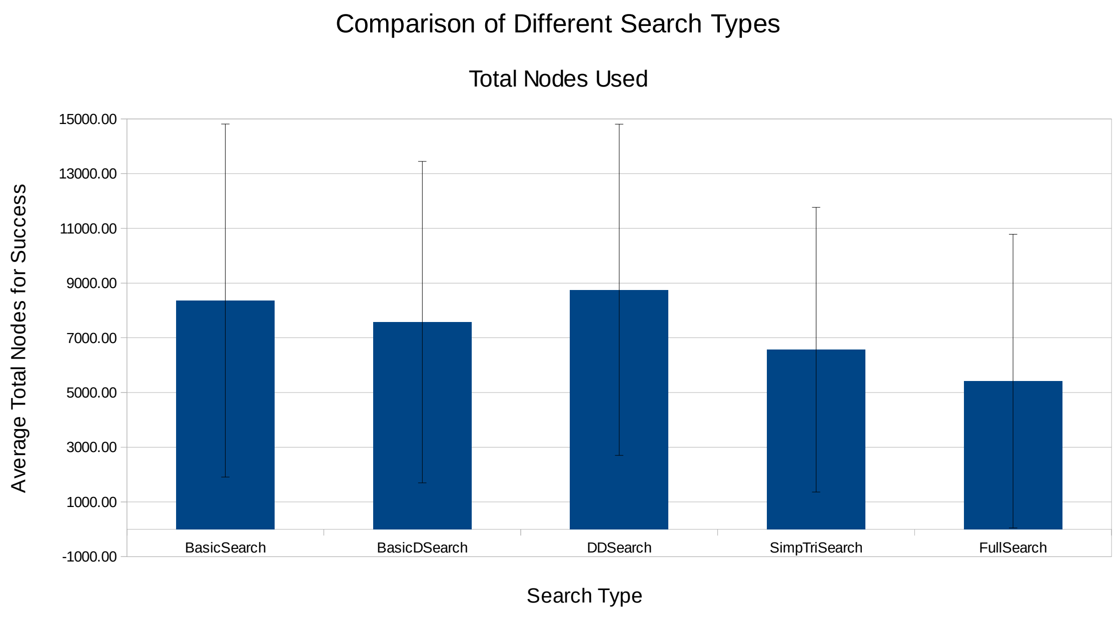


Figure 4.13 Average Searches Used in Different Searches

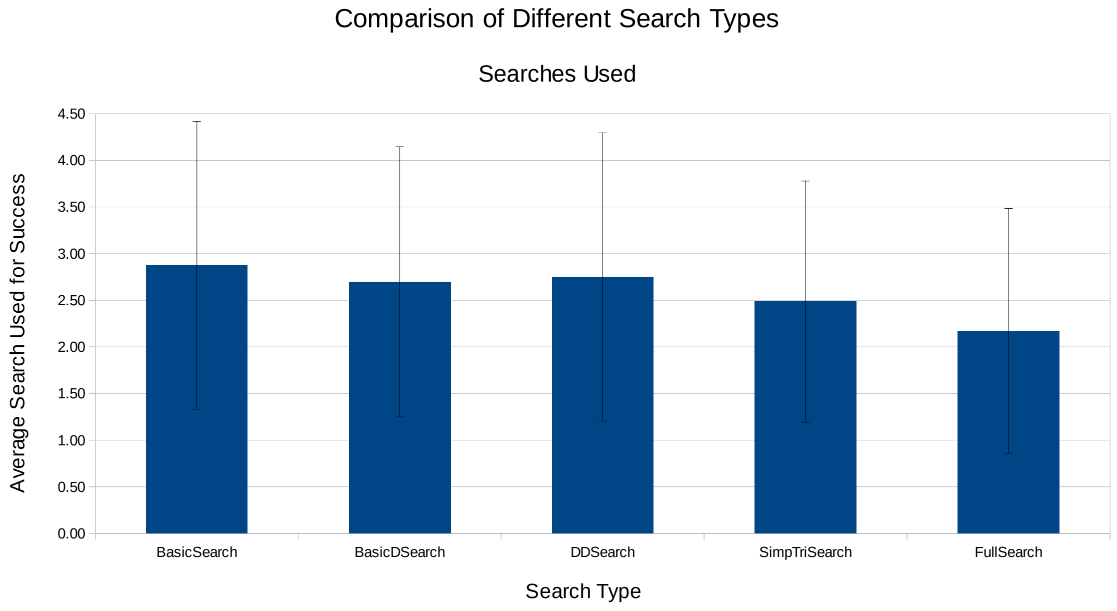


Figure 4.14 Average Success Times in Different Searches

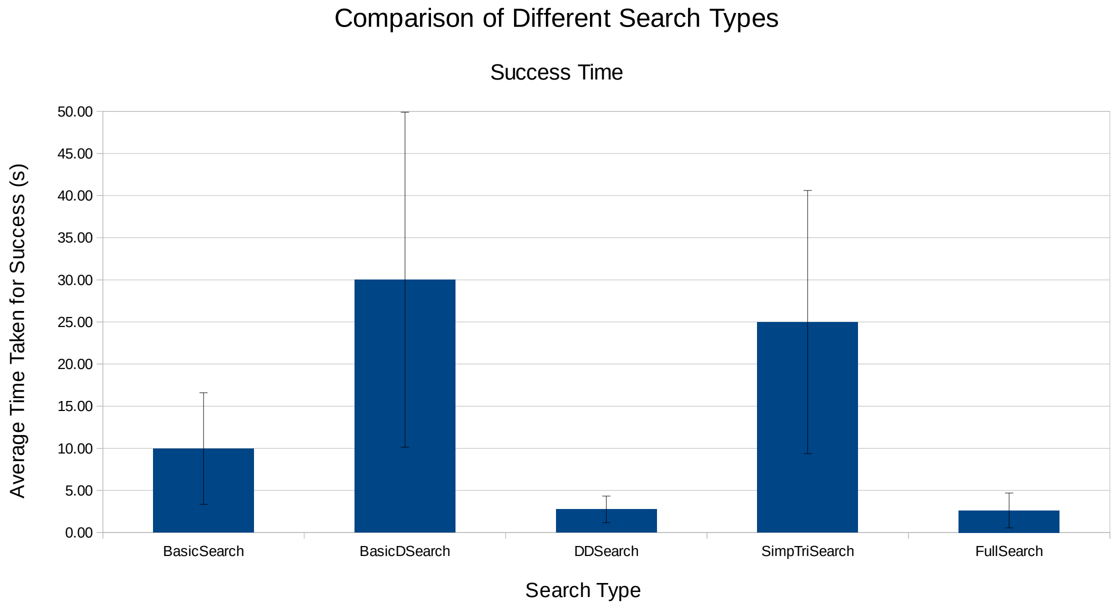
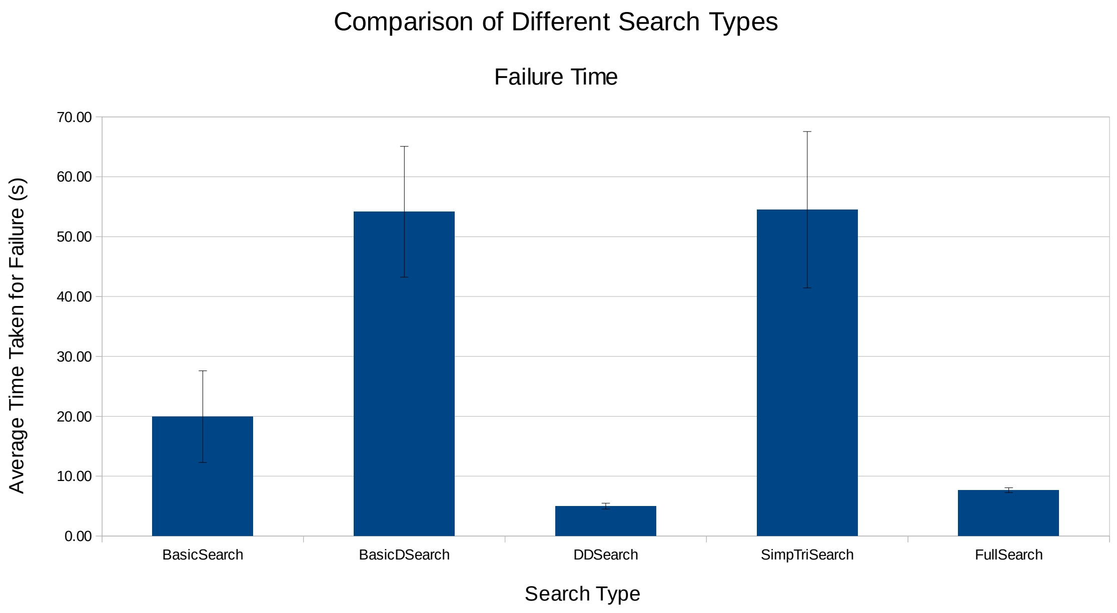


Figure 4.15 Average Failure Times in Different Searches



Effects on Success Rate

Distraction Stacks: The relative success rates can be seen graphically in figure 4.11. The basic search started with a low success rate at 16%, and the prevention of distraction stacks alone caused a huge increase to 53%. This was a much greater increase than expected, but it makes sense why it happened. The distraction stacks were not only using up a large portion of the nodes for the search, but were also creating an undesirable configuration of the tree which reduced the number of different times that the distraction was activated. Thus, when the search was able to perform in the desired fashion, the success rate increased dramatically.

Distance Constraint: In contrast to our success with preventing distraction stacks, the inclusion of the distance constraint actually decreased the success rate to 12%, lower than the initial value, despite the fact that the distraction stacks were still being prevented. The possible explanation for this is that one of the main ways that the RRT would be able move past difficult parts of the level is by selecting a lucky point further away, and the need to pick several points from within the difficult region in order to leave it decreased the success rate. It may also be possible that a different distance value would increase the success rate with this method.

Triangulation: When the triangulation was introduced, it also increased the success rate of the search. This was to be expected based on the previous experiment which showed that the majority of the nodes (56%) selected were in fact within obstacles and thus unreachable. Now, since a much larger portion of the nodes were viable, the success rate of the search improved. The success rate increased to 70% which is a significant increase, but more improvement could still be made.

Triangle Distance: The final change was the introduction of the distance function within the triangulation. In contrast to the introduction of the distance function to the non-triangulation version, this did in fact increase the success rate of the search, letting it reach 87% on this relatively difficult test level. Thus, it can be seen that by introducing changes to the RRT process to bias it to search in more desirable places, the success rate of the search was increased from a measly 16% to a respectable 87%. Here we see a nice synergy in our techniques—a distance constraint should reduce wasted samples that are taken too far away from the main search tree, but this needs to be combined with a technique like triangulation to ensure the constrained sample space is not saturated by infeasible obstacle areas.

Effects on Other Metrics

A number of other interesting characteristics were measured for these searches. These included the average number of searches used in a successful run (out of a maximum of five), the average number of nodes used summed over all searches used in a successful run (out of a maximum of 4000 per run, and thus 20 000 total), the average amount of time taken to find a solution, and the average amount of time taken to fail.

Nodes and Searches: The number of nodes used can be seen in figure 4.12, while the number of searches used can be seen in figure 4.13. For all of the search types, the number of searches used for success was between two and three, and it only appeared to be significantly lower for the full search, at 2.17 searches on average. However, the standard deviation within each type of search was significantly higher than the differences between the searches as the standard deviations were all approximately 1.4 searches. Thus, the significance of the differences between them is minimal. The results for total nodes used were similarly highly variable, with the difference between searches being much smaller than the difference in the averages.

Success Time: In contrast, the time taken for searches actually had significant differences. The time taken to succeed is seen in figure 4.14, while the time taken to fail can be seen in figure 4.15. The fastest success times were seen for the full search and the basic with distraction stack prevention and distance metric, with 2.62 and 2.75 seconds on average respectively. They also had relatively low standard deviations of 2.07 and 1.57 respectively. From this, it can be seen that when they succeeded, both of these searches did so quite quickly, but the difference in search time between them was not significant. The next fastest, was the basic search at 9.96 (+/- 6.62) seconds, and the slow searches were basic with distraction stack prevention, and triangulation without distance metric at 30.02(+/- 19.88) and 24.99(+/- 15.63) seconds respectively.

Failure Metrics: For the failed trials, calculating the number of searches used or the number of nodes used serves no purpose, as those would always be the maximum number of nodes or searches as determined by the parameters. Measuring the amount of time taken to fail however does give important information, and it should have much less noise than the success data. For all of the cases, the time to fail was approximately double that of the average success time for the same searches. The error associated with these averaged values were also smaller, but still relatively large for all cases that did not use the distance metric. From this we can hypothesize that far away nodes take more time to process than closer nodes. Our search process is partly dependent on the speed of Unity's linecasting primitives, which while proprietary likely take advantage of spatial partitioning of some form, and thus may be affected by distance in terms of number of object collisions they need to evaluate.

Overall Results

From this data, we can determine that preventing distraction stacks significantly increases the time taken by the search, but also significantly increases the success rate. The addition of a distance metric significantly decrease the run-time in all cases, although it only increased the success rate for the triangulated case. The triangulation itself has no significant effect on the run-time of the search, but it does give significant improvements in terms of success rate. In general, and as expected for a randomized search process like RRT, the results are highly variable in terms of time taken and resources used.

4.2.4 Motivation for New Searches

In full games, a player will be presented with multiple stealth puzzles. Although modeling such a scenario from an actual game is difficult, as the stealth events tend to be separated by other game elements, such as combat, resource acquisition and restoration, and non-trivial pathfinding, we can still experiment with a synthetic benchmark created by composing multiple instances of stealth puzzles, as an essential, if still incomplete part of analyzing a complete level. For this we make use of a compositional version of the search. We first do one experiment to show the reasons for developing a compositional search in the first place, showing its advantages over the simple non-compositional search. Then, we move on and show the advantages of the multiple midpoint search over the single midpoint search in its increased flexibility and coverage of different scenarios.

Midpoint vs Normal Search

Our design for a compositional search was motivated by results on the more complicated Two-Part Level. Initial experiments were run doing 100 trials of each search, using 4000 nodes and five search attempts. Unfortunately, the normal search, even with distraction stacks, triangulation, and distance constraints (the FullSearch) was unable to solve the level at all. Introducing the midpoint, compositional approach detailed in section 3.4.4 was more successful, able to compute a solution 28% of the time. This is still not a high success rate, but it nevertheless shows that a compositional version of the search allows the search to solve more complicated levels which it had previously been unable to solve.

The results of the midpoint search can be seen in table 4.3. As previously discussed, searches represents the average number of searches used, and nodes represents the total number of nodes used per trial on average. Here we also have SEARCHES2 and NODES2, which represent the same values but for the second half of the search. Since we do one search to reach the midpoint, then another to try to reach the end, we recorded information about the two halves separately. We calculate separate averages for successful trials and unsuccessful trials. We can also see that in both successful and unsuccessful trials more resources were used in the second half of the search than in the first half. This is likely due to the fact that for the second half of the search, it starts in the middle of the level and then searches in both direction simultaneously. Thus, less of the search resources are used effectively for the second half of the search. We do not restrict the search as in general a solution may involve backtracking or complicated paths, although for some levels it may be heuristically possible to improve success by biasing the geometric positions sampled in the search toward the goal position. The success rate could of course also be increased by giving the search more resources.

Table 4.3 Search Results for Two-Part Level.

Results of Search						
Name	Success	Searches	Nodes	Searches2	Nodes2	Time(s)
Success Vals	0.28	2.18	5540	2.93	8671	301.66
Std Dev		1.28	5378	1.46	6233	158.58
Name		Searches	Nodes	Searches2	Nodes2	Time(s)
Failure Vals		2.79	8674	3.96	15852	474.65
Std Dev		1.63	7623	2.04	8189	156.24

Multiple Midpoints vs Single Midpoint

Actual games do not always have stealth puzzles equally spaced throughout the level. A single choice of midpoint may thus be inadequate, and in general we cannot assume that the triangle closest to the middle is necessarily part of an actual solution path. As a final set of experiments, we thus evaluate our design for extending compositional search to situations where there are multiple geometric paths to the end, and in particular when the triangle closest to the middle is not necessarily part of an actual solution path.

Our approach in these cases is to find all midpoints, and do a search for each one. In this way it should be able to find a solution as long as one of the geometric paths actually leads to the end. Note that we focus on geometric midpoints as a solution path must pass

Figure 4.16 Single vs Multiple Midpoints

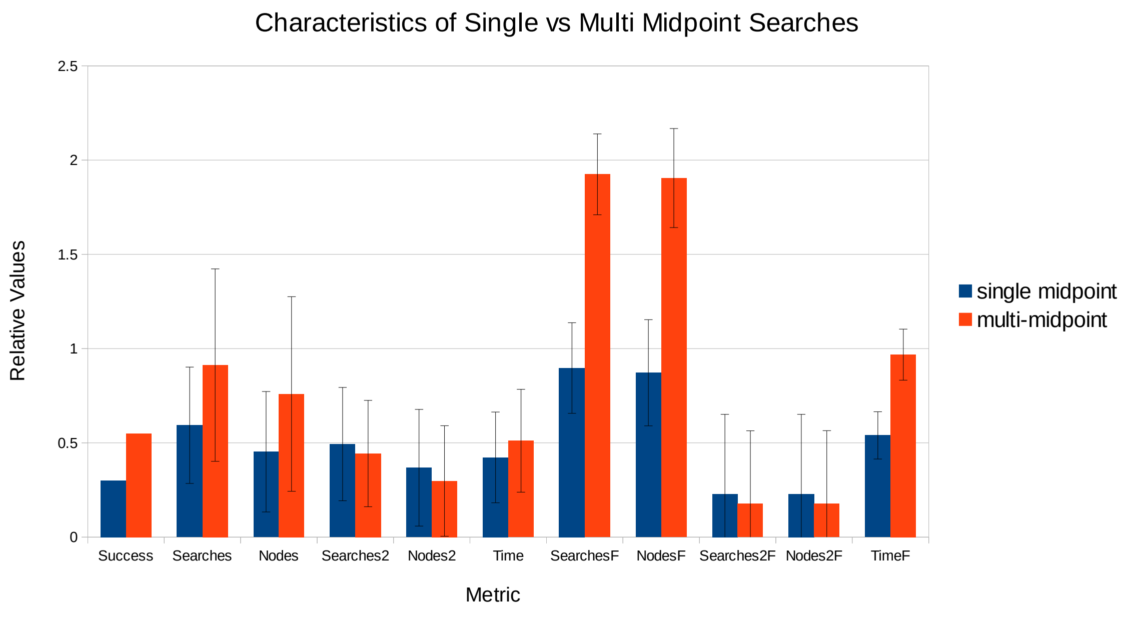


Table 4.4 Search Results for Two-Part Choice Level. Note: Succ. is used as a contraction for success

Results of Different Searches							
Single Midpoint	Name	Succ.	Searches	Nodes	Searches2	Nodes2	Time(s)
	Succ. Vals	0.30	2.97	9062	2.47	7361	126.90
	Std Dev		1.54	6395	1.50	6189	72.26
	Name		Searches	Nodes	Searches2	Nodes2	Time(s)
	Failure Vals		4.49	17443	1.14	4573	162.04
	Std Dev		1.20	5634	2.11	8461	37.64
Multiple Midpoint	Name	Succ.	Searches	Nodes	Searches2	Nodes2	Time(s)
	Succ. Vals	0.55	4.56	15192	2.22	5954	153.31
	Std Dev		2.55	10322	1.41	5864	81.86
	Name		Searches	Nodes	Searches2	Nodes2	Time(s)
	Fail Vals		9.62	38098	0.89	3557	290.49
	Std Dev		1.07	5260	1.93	7735	40.58

through one of them eventually, but this design naturally extends to choosing 'midpoints' arbitrarily spaced throughout the level, or in different dimensions.

To demonstrate our approach and evaluate how well it works, a level was created with two possible paths to the end geometrically, but one of them was blocked by a guard. This is the Two-Part Choice Level. In this case, the single midpoint search was run and compared to the multiple midpoint search as a motivating example for the uses of the multiple midpoint search. Again, there were 100 trials run and the parameters for both searches were five search attempts and 4000 nodes. Table 4.4 shows the numerical results. As previously discussed, searches represents the average number of searches used, and nodes represents the total number of nodes used per trial on average with SEARCHES2 and NODES2 representing the same values but for the second half of the search. We calculate separate averages for successful trials and unsuccessful trials. Additionally, in the multiple midpoint method data, for the numbers of average searches and average nodes, we took the sum across all midpoints. So, if a trial did five searches for one midpoint and three for a second midpoint, the number of searches used would be eight.

A graphical version of the data can be seen in figure 4.16. In order to show all the information on the same graph, this data is normalized by the maximum value possible for any search. Thus, SEARCHES were divided by five, and nodes were divided by 20 000. Time values were divided by 300 as this was a round number greater than the maximum time displayed for the graph. Bar-sets named with 'F's at the end indicate that the values are averaged for the failure trials, while those without are for successful trials. It can be seen that in some cases the Multiple Midpoint method has values greater than one. This is due to there being two midpoints, so it could use up to twice as many resources, the maximum allowable value per midpoint. If there were three possible midpoints instead of two, the worst case runtime would have been three times that of the single midpoint search, and this extends to any number of midpoints. However, if the level were subdivided into more than two sections this could increase the search time exponentially, as each time we have a choice of midpoints (or waypoints if the level is divided into more than two parts) it is in essence a branch point for the search.

The multiple midpoint search had a much higher success rate than the single midpoint search at 55% versus 30%, which illustrates its value. The single midpoint method still has a significant success rate because in this case because the two midpoints are at the exact same distance from the end. Thus, the single midpoint search chooses each of them with equal probability, and it is unsurprising that the single midpoint approach has a success rate close to half of that of the multiple midpoint. A doubling in success rate in the multiple midpoint case is paid for by a corresponding increase in worst-case resource cost. Since there are two possible midpoints, the multiple midpoint can do twice as much work in the worst case, if it fails for both midpoints. This can be seen in the time taken for failure, as well as the number of searches and nodes used for the first half of the search, as they are approximately double of those of the single midpoint search. It is interesting to note however that the time take to find a successful path is relatively equal, especially when accounting for the high variability of values. It can be seen that the both the time taken, and the resources used for successful trials is much less than that of failed trials. This indicates that the solution is often found early or not at all.

Chapter 5
Conclusions and Future Work

5.1 Conclusion

Stealth games are a popular genre of modern video games that exist both in pure forms as well as in conjunction with other genres. They rely on a complex interplay of different factors that either help or hinder the player in their goal to reach the end of the level undetected. Balancing these different factors in a difficult task for game developers and they benefit from any tools that make this process easier, like the tool created here for finding different solutions to stealth game levels.

One of the main factors that can be used in a stealth game is the ability of the player to affect the behavior of their enemies in different ways, most of which can be defined as different forms of distractions. In this work we have developed a new representation of guard movements that allows their distraction at any time and incorporated it into an RRT based algorithm for searching stealth game levels. We then proceeded to introduce numerous improvements to greatly increase the success rate of this search process and greatly decrease the search time. This algorithm was then extended into a compositional form that allowed it to solve much more complicated levels. The results of these improvements were shown on a variety of test levels, starting with simple levels, and proceeding to levels of increasing size and complexity which showcased the magnitude of the improvements and the flexibility of both the distraction representation and of the search itself. This algorithm was then additionally verified by testing on levels from existing games. This whole design is implemented entirely in the Unity3D game development framework showing its immediate applicability to the game development industry.

5.2 Future Work

A possible improvement lies in increasing the success rate of the hierarchical searches. A possible way to do this is for the second half of the search to bias the search to sample more nodes in the direction of the end goal as opposed to space that is in the opposite direction. However, that space also may need to be searched for more complicated levels. Additionally, a possible improvement lies in finding a way to determine a better way to find a midpoint. Ideally, this would be to find dangerous and safe locations and have waypoints located just past every large dangerous zone. Another way to choose waypoints would be to analyze the geometric path structure and place them at branch points. Additionally, the scalability of this to more waypoints would need to be tested, as with the current method of using ‘midpoints’ it appears that the worst case runtime could increase exponentially with the number of sections the level is divided into.

Another possible improvement is developing a more complicated methodology for choosing when and where to use distractions. While the current model allows for a lot of flexibility with regards to how distractions are used, the choice of them being used is done in a random manner. Some investigation of higher level planning for selection of distractions could be beneficial to the search.

A different place for a possible improvement would be in testing whether the player has been seen by a guard. Instead of testing at almost every time frame for intersection of the player with the guards field of view a faster method could involve computing the intersection of the line segment that is the player’s movement with the shape created by projecting the guard’s field of view through time. Although, this cannot be done in advance for all of the guards patrol, since the guards movement can change, it may be possible to compute this for the movement segments between waypoints or portions of those segments.

A final source of improvement would be to develop a more extensive test suite by performing a deep search of existing stealth games and extracting the levels therein. It would be beneficial to include both levels where stealth is intended to be the only solution as well as game levels from games that include other mechanics, such as combat, but where a stealth solution is still possible.

Bibliography

[1] Aaron William Bauer and Zoran Popovic. RRT-based game level analysis, visualization, and visual refinement. In *Artificial Intelligence for Interactive Digital Entertainment Conference*, 2012.

[2] S. Berchtold, D.A. Keim, and H.P. Kriegel. An index structure for high-dimensional data. *Readings in multimedia computing and networking*, page 451, 2001.

[3] Giant Bomb. Enemy distraction. `http://www.giantbomb.com/enemy-distraction/3015-5692/games/`. Accessed: 2016-04-02.

[4] Centerstrain01. Thief: Stealth walkthrough - master - ghost - part 26 - chapter 7 - the hidden city 1/2. `https://www.youtube.com/watch?v=icyKiYFWbBE`. Accessed: 2016-05-02.

[5] Brian Crecente. Volume is a pure stealth game with a slick aesthetic. *Polygon*, 2015.

[6] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*, chapter 9, pages 191–218. Springer-Verlag, 2008.

[7] Boris Delaunay. Sur la sphere vide. *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk*, 7(793-800):1–2, 1934.

[8] Clara Fernández-Vara. Game spaces speak volumes: Indexical storytelling. In *Digital Games Research Association*, 2011.

[9] GamerU. The evil within - how to distract enemies and kill them stealthily. `https://www.youtube.com/watch?v=rX4WMezYW0A`. Accessed: 2016-05-02.

[10] GruntShieldInc. State of decay - distraction. `https://www.youtube.com/watch?v=7pApHmrG6LY`. Accessed: 2016-04-02.

[11] Javed Hossain. Rapidly-exploring random tree (RRT). `https://en.wikipedia.org/wiki/File:Rapidly-exploring_Random_Tree_%28RRT%29_500x373.gif`, 2012.

[12] Damián Isla. Third eye crime: Building a stealth game around occupancy maps. In *Artificial Intelligence and Interactive Digital Entertainment*, 2013.

[13] Farzana Islam, Jauwairia Nasir, Usman Malik, Yasar Ayaz, and Osman Hasan. Rrt*-smart: Rapid convergence implementation of rrt* towards optimal solution. In *Mechatronics and Automation (ICMA), 2012 International Conference on*, pages 1651–1656. IEEE, 2012.

[14] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.

[15] Svetlana Kirillova, Juan Cortés, Alin Stefaniu, and Thierry Siméon. An NMA-guided path planning approach for computing large-amplitude conformational changes in proteins. *Proteins: Structure, Function, and Bioinformatics*, 70(1):131–143, 2008.

[16] James J. Kuffner and Steven M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 995–1001. IEEE, 2000.

[17] J.J. Kuffner and S.M. LaValle. Randomized kinodynamic planning. In *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, pages 473–479, 1999.

[18] Mark J. Nelson. Game metrics without players: Strategies for understanding game artifacts. In *Artificial Intelligence in the Game Design Process*, 2011.

[19] Luke Perkins. Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition. *Artificial Intelligence and Interactive Digital Entertainment*, 10:168–173, 2010.

[20] David Pizzi, Jean-Luc Lugrin, Alex Whittaker, and Marc Cavazza. Automatic generation of game level solutions as storyboards. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(3):149–161, 2010.

[21] Tommy Reddad and Clark Verbrugge. Geometric analysis of maps in real-time strategy games: Measuring map quality in a competitive setting. Technical Report GR@M-TR-2012-3, GR@M: Games Research At McGill, School of Computer Science, McGill University, sep 2012.

[22] Yinxuan Shi and Roger Crawfis. Optimal cover placement against static enemy positions. In *Foundations of Digital Games*, pages 109–116, 2013.

[23] Alexander Shkolnik and Russ Tedrake. Path planning in 1000+ dimensions using a task-space Voronoi bias. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 2061–2067. IEEE, 2009.

[24] Adam M. Smith. Open problem: Reusable gameplay trace samplers. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[25] R. Smith. Level-building for stealth game-play - game developer conference. `http://www.roningamedeveloper.com/Materials/RandySmith_GDC_2006.ppt`, 2006.

[26] Jonathan Tremblay, Alexander Borodovski, and Clark Verbrugge. I can jump! exploring search algorithms for simulating platformer players. In *Experimental AI in Games Workshop (EXAG 2014)*, October 2014.

[27] Jonathan Tremblay, Pedro Andrade Torres, Nir Rikovitch, and Clark Verbrugge. An exploration tool for predicting stealthy behaviour. In *The Second Workshop on Artificial Intelligence in the Game Design Process*, 2013.

[28] Jonathan Tremblay, Pedro Andrade Torres, and Clark Verbrugge. An algorithmic approach to analyzing combat and stealth games. In *2014 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, August 2014.

[29] Jonathan Tremblay, Pedro Andrade Torres, and Clark Verbrugge. Measuring risk in stealth games. In *FDG'14: Proceedings of the 9th International Conference on Foundations of Digital Games*, April 2014.

[30] TV Tropes. Throwing the distraction. `http://tvtropes.org/pmwiki/pmwiki.php/Main/ThrowingTheDistraction`. Accessed: 2016-04-02.

[31] Qihan Xu, Jonathan Tremblay, and Clark Verbrugge. Procedural guard placement for stealth games. In *Fifth Workshop on Procedural Content Generation in Games (PCG 2014)*, April 2014.

[32] Andrew Yoder. police-station. `https://mclogeblog.wordpress.com/2015/09/11/on-thiefs-level-design-maps-and-territories/`.

[33] W. Zeng and R.L. Church. Finding shortest paths on real road networks: the case for A*. *International Journal of Geographical Information Science*, 23(4):531–543, 2009.

[34] Liangjun Zhang and Dinesh Manocha. An efficient retraction-based RRT planner. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 3743–3750. IEEE, 2008.