Escape the Sensors

Rio Carasco

MATH5872M

Supervisor: Peter Gracar

Date: 14/08/2023

Abstract

Path finding is a complex task, navigating complex terrains—marked by unpredictable topographies and potential threats—poses a significant challenge. While traditional algorithms are suited to structured environments, they often stumble when confronted with the reality of multi-dimensional landscapes in the real world. The research conducted in this paper pioneers a unique approach to bridge this divide, creating and illustrating the steps behind a novel algorithm adept for both 2D and 3D terrains. Simulated annealing lies at the heart of this algorithm, conducting a probabilistic method with the ability to approximate a function's global minimum. Primarily designed to find paths that minimise detection from sensors, where both time and proximity play crucial roles in detection, this study further adapts the model to represent tangible hazards, ensuring the paths discerned are not only optimal but also realistic.

To validate the algorithm's efficiency and efficacy, a series of data analyses were conducted. By tweaking the parameters of the simulated annealer, the research presents a method of finding optimal parameters for the algorithm to reach an optimal path most efficiently. Additionally, we will explore how adjusting the cost function changes how the optimal path is built and what features are more strongly favoured. The findings reveal significant shifts in path optimization, underscoring the algorithm's sensitivity to parameter changes and its potential for fine-tuning in relation to modelling real life scenarios.

Coupling theory with practicality, the research also explores a real-world application—mountain rescues. Scenarios like this are known for their urgency and dangerous terrains involved. Which will serve as a testament to the algorithm's potential. Simulations of these high-stakes situations emphasise the algorithms potential to revolutionise rescue operations and potentially save lives.

In essence, this study not only delves into a new direction in terrain path finding but also emphasises the real-world implications. Through rigorous testing, data analysis, and realistic construction of simulations, the research offers a comprehensive exploration of the algorithm's capabilities, setting the stage for future enhancements and broader applications.

Contents

1 Introduction

In the vast expanse of our 3D world, generally the shortest distance between two points is a straight line. But what if that line traverses diverse topological landscapes, avoids sensors, and is influenced by factors from gradient to speed? This is the intricate realm of modern path finding.

Transitioning from traditional algorithms working from structured grids to the unpredictable landscapes of the real world, the challenges multiply. The randomness of natural landscapes, combined with factors like terrain gradient, sensor positioning, detection probabilities, and speed, necessitates a more nuanced approach. This has transformed path finding from a straightforward computational task to a complex simulation of reality.

This research aims to introduce a novel algorithm tailored for both 2D and 3D environments. The goal is realism, the model's adaptability is crucial in making it a potential tool for real-world scenarios.

Central to this research are the following inquiries:

- How can the complexities of real-world terrains be effectively modeled?
- How can we realistically represent various factors, from terrain gradient to sensor strength?
- How can the model be adapted and fine-tuned to represent different real-world challenges?
- How does the tweaking of parameters within the simulated annealing algorithm influence the efficiency in achieving the lowest cost path?

Post this introduction, a comprehensive literature review is undertaken, tracing the history and evolution of path finding. It focuses on the success of early algorithms and how they have adapted over time. This will be influential in inspiring the methods for my own algorithm and setting the stage for path finding.

We will then appropriately define the basis of the problem at hand and how we are going to tackle it. We are going to collect the tools that are available to us and use them appropriately. Understanding the complexities that realism brings and how we can use mathematics to model it. Addressing challenges that we will have to overcome too.

Subsequent sections dissect the proposed algorithm explaining the fundamentals and mathematical rigour behind it's workings. We delve into the inner workings of the simulated annealer and how we can modify it to our benefit. The code that makes up the 2D algorithm, the basis of the entire algorithm, is shared and broken down to be easily understood. This is then built upon in the 3D model with some extra functions that are critical for emulating 3D environemnts.

The theory is then put to the test in the analysis section with practical simulations. The goal is to tweak the simulated annealer parameters to find the parameters with the most impact and the range of values that are optimal. Next, focusing on the cost function and how it affects the outcome of the path on the same terrain. Further demonstrating the broad capabilities of the algorithm.

It will then be time to emulate a real-world scenario. Real-world data serves as the crucible, testing the model's robustness and adaptability. We will construct a model catering to all the needs of the scenario at hand, showcasing the effectiveness of the algorithm.

The implications of this research are profound. From guiding rescue teams across mountain ranges to navigating drones in sensor-laden environments, the potential applications are vast and transformative, heralding a new era in path finding.

Crafting a path finding model that mirrors the complexities of the real world, is the primary objective in this research. By integrating diverse factors and testing against real-world data, the research seeks to offer a tool that's not just efficient, but also deeply attuned to real-world challenges.

2 Literature Review

2.1 Introduction to the Literature Review

Path finding has become a cornerstone in various computational and technological domains. At its essence, path finding determines the most optimal trajectory between two distinct points. While this might seem straightforward, the computational intricacies and challenges it presents are plenty, especially when the terrains and scenarios in question are complex.

The literature on pathfinding is vast, reflecting its importance across various applications, from robotics to video games, urban planning, and even biological simulations. As our understanding of complex systems has deepened, so too has the sophistication and capability of pathfinding algorithms.

This literature review will provide a comprehensive overview of the pathfinding domain. The historical evolution, challenges, and highlighting the innovative solutions that have been proposed over the years. We will explore traditional pathfinding algorithms, their strengths and limitations, especially when confronted with realism.

The review will also introduce the reader to the principle of simulated annealing, the history, how it has been used in the past, and how we can use it for our benefit.

As we are working through the literature, it is important to keep in mind that the research over viewed will play a strong role in the direction of the algorithm we develop. Guiding the trajectory of the research endeavors and shaping the innovative solutions we seek to develop. Deriving inspiration from the methods discussed to aid in the development of the algorithm.

2.2 Historical Overview of Path finding

The computational aspect of path finding, where algorithms and mathematical models are employed to determine the most efficient route, is a relatively modern development. Closely tied to the rise of computer science and operations research in the 20th century.

2.2.1 Early Beginnings and Classical Algorithms

It is said that the formal study of path finding in computational terms began with the introduction of graph theory in the 18th century. Leonhard Euler laid the foundation for representing complex networks, such as roads or pathways, as graphs. This representation allowed for the development of the first algorithms to find the shortest path between two points.

Dijkstra's Algorithm Edsger Dijkstra the founder of Dijkstra's Algorithm formulated in 1956, stands as one of the earliest and most fundamental path finding algorithms [5]. It proves it's worth with it being the backbone of every navigation system, including Google Maps [5].

Dijkstra's Algorithm has been effectively used in structured environments like city grids. It operates on a graph defined by nodes and arcs, where nodes are the vertices, and the arcs are the path between the nodes. The paths are weighted according to the scenario at hand and the algorithm calculates the shortest path from the starting node to the finishing node [5].

It has furthered the navigation system by suggesting alternative routes to avoid traffic and delays, and even providing visual previews of the route to be traveled.

However, when it comes to applying Dijkstra's Algorithm to complex terrains the limitations arise. Despite its widespread application, the algorithm only works in cases where the shortest path is based on weighted paths and non-negative lenths[5]. The algorithm falls short when dealing with more complex terrains where factors like gradient resistance, speed reduction due to terrain, and varying speeds complicate the model.

Strides have since been made to extend upon the existing and overcome some of its limitations, especially considering negative weights. Noto and Sato introduced a method for the shortest path search by extending the Dijkstra algorithm, allowing for more flexibility in handling complex scenarios [6]. Effectively enhancing the applicability of Dijkstra's Algorithm in various domains.

2.2.2 The Rise of Simulated Annealing

While Dijkstra's and other deterministic algorithms proved useful among many applications, they fell short when dealing with complex landscapes with numerous local optima. This such problem led to the development of probabilistic techniques.

The simulated annealing algorithm was originally inspired by the annealing process in metallurgy. It was introduced in the 1980s as a probabilistic method to approximate the global optimum of a given function. The general pretense was to allow the algorithm to "explore" the solution space, occasionally accepting worse solutions to escape local optima. Overtime the likelihood of such acceptance decreasing, mimicking the cooling of metal [3]. The genesis of the simulated annealing algorithm and evolution for solving difficult optimization problems is well described in [3].

2.2.3 Local Minimum Optimization and Realism in Models

As the field matured, researchers recognized the importance of escaping local minima and the challenges it presented. This was especially true in terrains with numerous valleys and peaks. Algorithms that could identify and escape these local minima became crucial, as is demonstrated in [15]. Techniques described in [15] like hill climbing, genetic algorithms, and particle swarm optimization were developed to address these challenges. Each bringing its unique approach to the problem.

Path finding models have been consistently driven towards realism over the decades. Early models while effective, often operated on simplified representations of terrains. As the ability to model real-world complexities and computational power increased, models began to incorporate more realistic features. Factors include, varying elevations, unpredictable obstacles, dynamic environments, and even the behavior of agents moving through terrain. Allowing models to become more reflective of real life scenarios.

From classical algorithms to the incorporation of probabilistic techniques and the drive for realism, the field has seen tremendous growth and evolution. As we look to the future, the lessons from the past will undoubtedly continue to guide and inspire new breakthroughs, pushing the boundaries of what's possible in path finding and optimisation.

The field of path finding through the ages, from classical algorithms, to the incorporation of probabilistic techniques has seen tremendous growth and evolution. The quest for realism continues and the techniques and inspiration from the classical algorithms still hold a firm foundation for modern development of path finding algorithms.

2.3 Path finding in Complex Terrains

As covered already creating path finding models that accurately depict the intricacies of the natural world are a challenging endeavour. Structured environments provide a simple form of representation which can't be said for natural realistic terrains. They demand a more nuanced approach which we will look at now.

Challenges Posed by Natural Terrains:

Varying Elevations and Gradient Resistance: One of the primary challenges in three dimensional natural terrains is the presence of varying elevations and the concept of gradient resistance. As one moves uphill, the resistance or energy required increases. Conversely, moving downhill might be faster but can also pose risks, especially in steep descents. Modern algorithms have been adapted to factor in gradient resistance by measuring energy expenditure as a variable concerning a path [9].

Terrain-Induced Speed Reduction: Marshy lands, sandy deserts, dense forests, or rocky terrains can significantly reduce speed. Addressing the speed reductions caused by different terrains, means algorithms now incorporate terrain cost functions. These functions assign a "cost" to each type of terrain based on its difficulty or vary the speed factor of the cost function accordingly [9].

Speed Varying with Conditions: Energy conservation is a necessary consideration when dealing with path finding. With speed and energy expenditure being intimately correlated speed becomes an important factor in path determination. Other factors correlated include gradient, terrain and mode of transport. Recognising that speed can vary, advanced path finding systems can estimate energy expenditure, adjusting traversal speeds based on the factors discussed. This ensures that the path calculated is the most optimal given the prevailing energy considerations. [9]

Dynamic Environmental Factors: Natural terrains are not static. Dynamic environmental factors, from changing weather conditions like rain or snow to natural disasters like landslides or avalanches, is another consideration for models emulating the real-world. Given the dynamic nature of natural terrains, modern algorithms are designed to be adaptive. Some modern algorithms incorporate methods to recalculate paths on-the-fly, adjusting to changes in the environment. This is especially crucial in scenarios where time is of the essence, like search and rescue operations. [9]

Complex Terrains The paper by Han et al. [9] presents a novel approach to autonomous path finding in three-dimensional curved surface terrain. For varying elevations and gradient resistance, the paper employs a 3D terrain-processing module that considers the height of the terrain as the cost incurred by the agent to traverse it. This factors in the resistance or effort required to move uphill or downhill for the agent.

Terrain-induced speed reduction is tackled through terrain rasterisation and conversion. Terrains like marshy lands, deserts, forests, or rocky areas are represented in a 3D raster map, allowing the algorithm to weigh paths by the effort required to traverse. Dynamic environmental factors are addressed through the retrospective-double deep Q-learning (R-DDQL) algorithm. This algortihm utilises global information and incorporates a retrospective mechanism to adapt to changes in the environment.
The proposed methods estimate energy expenditure, adjusting traversal speeds based on current conditions. Making the approach suitable for complex natural terrains and scenarios and a good example of an algorithm to derive ispiration from. [9]

2.4 Simulated Annealing in Pathfinding

Among path finding algorithms, simulated annealing has emerged as a particularly promising candidate. The algorithm offers a blend of flexibility, robustness, and adaptability. This section delves into the integration of simulated annealing into path finding, exploring its historical applications, benefits, challenges, and the nuances of parameter tuning.

The paper titled "A Simulated Annealing Algorithm and Grid Map-Based UAV Coverage Path Planning Method for 3D Reconstruction" [13] presents an approach to path planning for Unmanned Aerial Vehicles (UAVs) in the context of 3D terrain reconstruction. With respect to [13], the authors also highlight the flexibility of the algorithm. The fine-tuning of parameters like flight height and the field of view (FOV) of the camera ensure sufficient image overlap and energy efficiency. The authors employ a Simulated Annealing Algorithm (SAA) to calculate near-optimized paths for UAVs, adapting to the specific requirements of 3D terrain reconstruction. This paper serves as a demonstration of how useful the simulated annealing algorithm can be when integrated into path finding.

Simulated Annealing was chosen for its ability to find a global optimal solution. Compared to other algorithms, simulated annealing offers a balance between exploration and exploitation. Whilst other algorithms might converge faster to a solution this may be a local minima, avoiding local minima is a key feature of simulated annealing. The probabilistic nature of simulated annealing allows it to explore a broad solution space, making it suitable for complex terrains and the preferred choice for the authors in this context.

Over the years, numerous studies have sought to harness the power of simulated annealing for path finding. Early research focused on basic grid-based terrains like the structured environments that were utilised for classical algorithms. However, as the algorithm matured, its applications expanded to more complex environments, from urban landscapes to intricate natural terrains.

Benefits:

- **Adaptability:** Simulated annealing's probabilistic nature allows it to adapt to a wide range of terrains and scenarios. In the paper by Xiao et al. [13], the Simulated Annealing Algorithm (SAA) is utilized to calculate near-optimized paths for UAVs, adapting to the specific requirements of 3D terrain reconstruction.

- **Avoidance of Local Optima:** One of the standout advantages of simulated annealing in path finding is its ability to escape local optima, ensuring a more comprehensive exploration of possible paths. This is demonstrated in the referenced paper, where SAA helps in finding a global optimal solution rather than getting stuck in a local minimum [13].
- **Parameter Flexibility:** The algorithm's parameters can be fine-tuned to suit specific challenges, allowing for a tailored approach to different path finding problems. The paper illustrates how parameters like flight height and the field of view (FOV) of the camera can be adjusted to ensure sufficient image overlap and energy efficiency [13].

Challenges:

- **Parameter Sensitivity:** While the flexibility of simulated annealing's parameters is a strength, it can also be a double-edged sword. Incorrect parameter values can lead to suboptimal paths or excessive computation times. The referenced paper does not explicitly discuss this challenge, but it is a known issue in the application of simulated annealing.
- **Computational Intensity:** For very large terrains or highly detailed environments, simulated annealing can be computationally intensive. The paper by Xiao et al. does not delve into this challenge, but it is a consideration in the broader application of simulated annealing in pathfinding.

Conclusion The integration of simulated annealing into path finding represents a significant advancement in the field. The flexible, adaptable approach, that simulated annealing offers addresses many of the challenges posed by complex terrains and dynamic environments. The paper by Xiao et al. [13] serves as an example of how simulated annealing can be applied to achieve paths based on conditions and costs. The paper highlights both the benefits and potential challenges of this approach.

However, its success hinges on a deep understanding of its parameters and their implications. The fine-tuning of these parameters and the incorporation of real-world data promise to push the boundaries of what's possible in path finding.

2.5 Real-world Implications and Applications

Path finding applications permeate various sectors, from robotics to emergency response, and have tangible, sometimes life-saving, implications. With the integration of sophisticated algorithms, and the methods we have discussed, it has enabled path finding solutions to address some of the most pressing challenges in the real world.

Autonomous Vehicles: One of the most prominent applications of pathfinding is in the domain of self-driving cars. The goal is to autonomously navigate complex urban environments. These vehicles rely on a combination of sensors, maps, and real-time data to achieve this. Research has shown that advanced pathfinding algorithms can significantly reduce travel time, fuel consumption, and accidents.

In the context of Internet of Vehicles systems, observability is a challenging factor that affects the behavior of groups of autonomous vehicles. Observability is particularly limited on the occasions when communication is unstable, such as in crowded parking lots [10]. It is in these scenarios that vehicles must rely on local observations and cooperative behavior to ensure safe and efficient trips. This problem can be abstracted to multi-agent pathfinding. Agents confined to a graph must find collision-free paths to their goals, minimizing an objective function like travel time. Traditional algorithms often assume the existence of a central controller with full knowledge of the environment, making them unsuitable for partially-observable setups. The work by [10] introduces an approach that decomposes the problem into two sub-tasks: reaching the goal and avoiding collisions. Utilizing reinforcement learning methods such as Deep Monte Carlo Tree Search, Q-mixing networks, and policy gradients methods. The authors design policies that map agents' observations to actions. A policy-mixing mechanism is introduced to create a single hybrid policy that allows each agent to exhibit both individual behavior (reaching the goal) and cooperative behavior (avoiding collisions with other agents). Extensive empirical evaluation shows that the suggested hybrid-policy outperforms standalone state-of-the-art reinforcement learning methods for this kind of problem by a notable margin.

Pathfinding for Robotics and Video Games The paper by Algfoor et al. [1] provides a comprehensive overview of pathfinding algorithms and techniques related to applications in robotics and video games. The authors categorise path finding algorithms based on 2D/3D environment search and focus on the developments and improvements made in the last decade. The paper is divided into two main parts: graph generation and the pathfinding algorithm itself.

In the graph generation section, the authors discuss the terrain topology problem. This lays the foundation of pathfinding navigation in continuous environments. They explore different techniques for representing the navigation environment, such as skeletonization, which extracts a skeleton from the continuous environment. In addition, cell decomposition, which breaks down the traversable space into cells.

In the pathfinding algorithm section, the authors review various search algorithms used by games and robotics developers. These include heuristic functions and best-first search methods. They evaluate these algorithms based on factors such as execution time, memory overhead, and the environment's static, dynamic, or real-time nature.

The paper concludes with a look at pathfinding research in the field of video games. The authors' intention is to inspire researchers and developers and to shed light on the relationship between different types of terrain topologies and the potential for using pathfinding techniques in more extensive areas [1].

The Impact of Accurate Path finding in Critical Situations like Mountain Rescues In critical scenarios such as boat rescue operations, pathfinding plays a vital role in ensuring timely and efficient responses. Unmanned Aerial Vehicles (UAVs or drones) are employed to travel long distances over the sea to reach distressed boats. They utilise a grid of floating charging stations (CSs) and Base Stations function. The challenge lies in determining the "optimum path" from the BS to the boat and back. Typically the shortest path involving hops via CSs. However, the problem is my diverse, considering objectives for both drones and boats. This includes priority, the number of chargings for the UAV, and average waiting time for the boats. A heuristic extension called "red-gray path" has been proposed in this paper [4]. Providing savings in flight distance based on the boat's position within the CS grid. The design of the rescue infrastructure is highly dependent on the "drone range," or the maximum flight range achievable on a single battery charge. Additionally, the geometry of the CS grid significantly impacts the effectiveness of the heuristic. While a square grid offers better savings, a triangular grid provides better coverage with fewer CSs for the same mission area. This study illustrates the complexity and importance of pathfinding in life-saving missions, where the right algorithm can make a difference in both efficiency and effectiveness [4].

2.6 Recent Advancements and Future Directions

As computational capabilities expand and the demand for more efficient and realistic pathfinding solutions grows, researchers are continually pushing the boundaries of what's possible. This section delves into the latest advancements in the field and casts a gaze into the promising horizons of future research.

Hybrid Algorithms: One of the most notable trends is the fusion of simulated annealing with other optimization techniques. The combination of such algorithms such as, genetic algorithms or particle swarm optimization, leads to potentially offering faster convergence and better solutions.

Adaptive Simulated Annealing: Traditional simulated annealing relies on fixed parameters. However, adaptive variants dynamically adjust parameters like temperature or step size based on the current state of the solution. Adapting to more efficient exploration and exploitation.

Deep Learning and Path finding: With the rise of deep learning, neural networks are being integrated into path finding tasks. For instance, neural networks can be trained to predict terrain difficulty or sensor detection probabilities. Enhancing the realism and accuracy of path finding models.

Quantum Simulated Annealing: As quantum computing gains traction, there's growing interest in quantum variants of simulated annealing. The main advantage of these types of algorithms is to leverage quantum mechanics to explore multiple solutions simultaneously, potentially offering exponential speedups. A significant advancement in this field has been made by developing a quantum algorithm to solve combinatorial optimization problems. This is through quantum simulation of a classical annealing process [11]. An innovative approach that combines techniques from quantum walks, quantum phase estimation, and the quantum Zeno effect. It can be seen as a quantum analogue of the discrete-time Markov chain Monte Carlo implementation of classical simulated annealing. Remarkably, the scaling of the algorithm is with the inverse of the square root of the minimum spectral gap of the stochastic matrix used in the classical simulation. This results in the quantum algorithm outperforming the classical one, which scales with the inverse of the gap. These enhancements pave the way for more efficient optimization solutions, harnessing the unique capabilities of quantum computing.

2.6.1 Potential Areas of Future Research and Expected Advancements in the Field

Environmentally-aware Pathfinding: Future path finding algorithms might incorporate environmental considerations, such as minimising ecological impact or adhering to sustainable practices.

Integration with Augmented Reality (AR) and Virtual Reality (VR): As AR and VR technologies mature, pathfinding algorithms could pave the way for immersive and realistic navigation experiences in virtual worlds.

Personalised Pathfinding: Focusing on user data and preferences, future pathfinding solutions might offer personalised routes. Catering to individual needs or desires, be it scenic routes, areas of interest, or specific terrains.

Multi-Agent Pathfinding MAPF with continuous time represents a significant advancement in the field of path finding, particularly relevant to real-world applications such as warehouse management and autonomous vehicles [2]. Traditional MAPF algorithms often rely on discretised time steps, limiting their applicability in dynamic environments. The recent work by Andreychuk et al. introduces two novel algorithms, Continuous-time Conflict-Based Search (CCBS) and SMT-CCBS. These algorithms do not rely on time discretisation, allowing for more flexible and optimal solutions [2]. These algorithms build on existing techniques like Safe Interval Path Planning (SIPP) and Conflict-Based Search (CBS), but adapt them to handle continuous time and non-uniform action durations. The results show that both algorithms can efficiently solve non-trivial MAPF problems. Making them a promising candidate for future applications. The move towards continuous time in MAPF not only enhances the quality of solutions but also broadens the scope of problems that can be addressed. This will allow for more sophisticated and adaptive path finding systems in the future.

2.7 Conclusion of the Literature Review

The journey through the literature has provided a comprehensive overview of the intricate domain of pathfinding. The historical roots, the challenges posed by complex terrains, and the innovative solutions that have emerged over time. From the rudimentary beginnings of pathfinding algorithms to the sophisticated models of today.

Central to our exploration was the principle of simulated annealing, which has carved a niche for itself in the realm of optimisation. Its ability to escape local optima and approximate the global optimum of a function has made it a favored choice for many researchers. The integration of simulated annealing with path finding has opened up new avenues, offering solutions that are not only efficient but also adaptable.

The real-world implications of path finding cannot be overstated, especially in critical scenarios like mountain rescues. The literature is replete with case studies and research showing the impact of accurate path finding solutions. Most importantly the potential to save lives but also to optimize operations, and navigate challenging terrains with precision.

Looking into recent advancements the field is on the cusp of transformative innovations. From hybrid algorithms and adaptive simulated annealing to the integration of deep learning and quantum mechanics, the horizon is promising.

The literature serves as both a foundation and a beacon, guiding researchers like us as we navigate the intricate maze of challenges and opportunities that lie ahead.

3 Problem Description

3.0.1 Scenario Overview

The most general description of the problem is ”escaping the sensors”. While the primary function of sensors is to enhance security, there are legitimate scenarios where the need arises to bypass sensors without triggering an alarm. For instance, special operations forces may need to infiltrate a hostile area to rescue hostages. Wildlife researchers may need to access a restricted natural habitat without disturbing the local fauna.

In other such cases, the ”sensors” may not be man-made but are natural hazards like areas prone to avalanches, rockfalls, or unsteady ground. Accurate sensor data can provide invaluable information about these hazards, helping rescue teams navigate safely while reaching those in need. The challenge here is to bypass these natural "sensors" effectively to carry out rescue operations without endangering more lives.

3.0.2 Objective

The primary objective of this study is to develop a mathematical model that allows for the successful navigation through a sensor-laden environment. The model must be robust and flexible and reach a specific target with the smallest probability of detection. However, the goal is to find a path that is also realistic and practical. This means considering various constraints determined by the scenario at hand.

Moreover, the model aims to cater to specific requirements based on the objective of the mission. For example, a rescue operation may prioritize speed and safety, while a covert surveillance mission may prioritize minimal detection. Therefore, the model should be flexible enough to incorporate these varying requirements into its optimization framework.

3.0.3 Additional Considerations

An important aim of this study is to create a model that can be tailored to fit any scenario, making it highly adaptable and versatile. This includes any situation where navigation through sensor-monitored or hazardous areas is required. To achieve this level of adaptability, it is crucial to design the model in a way that allows for the easy inclusion of different variables and constraints. Furthermore, the model should be scalable, capable of handling a large number of sensors and agents. This enhanced focus on adaptability and realism ensures that the model is not just theoretically sound but also practically applicable in a wide range of real-world situations.

3.1 Mathematical Foundation

The foundation of our model is a Cartesian two-dimensional coordinate system that serves to map positions within the model. For the three-dimensional model a function/surface will act as the landscape upon which the path will traverse, modeled as a height map. In this height map, the height is given at every (x,y) position, providing a realistic representation of the terrain.

The primary inputs to the model include:

- The coordinates where each sensor is located.
- How many sensors are in the environment.
- The coordinates for the starting and ending points of the path.

The path itself will consist of a specified number of checkpoints, all of which will lie on the landscape created by the height map. These checkpoints will be connected by vectors, forming a continuous path from the start to the end point. This will simulate the movement of an "agent" along the path.

3.1.1 Sensors and Detection Probability

The model incorporates sensors whose detection capabilities are governed by the inverse square law. This mathematical formulation allows for a realistic representation of how the probability of detection decreases as one moves farther away from the sensor. The inverse square law can be adjusted to fit the specific characteristics of different types of sensors or environmental conditions. The total probability of detection from sensors at a point is simplified to the product of the two closest sensors.

$$p = \frac{1}{1 + \left(\frac{\text{dist}}{\text{sig}}\right)^2}$$

Line of Sight Resistance Factor
To enhance the model's realism, a resistance factor will be introduced. This factor adjusts the sensor's detection strength based on the degree of line of sight it holds with the agent.

Considering $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$ is coordinate location of the agent and $\begin{pmatrix} a \\ b \\ c \end{pmatrix}$ is the coorinate location of the sensor.

$$L = \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \lambda \begin{pmatrix} a - x \\ b - y \\ c - z \end{pmatrix}$$

$$Height(\alpha, \beta) = h$$

With a specified interval length a range of λ values are chosen between the sensor and agent. At each λ location the height of L is calculated and is compared to h, the height of the surface at that point. This is to find if the line of sight is broken. The percentage of λ locations with line of sight breaks will be proportional to the resistance.

Time Factor in Detection Probability
The time spent within the sensor's detection radius is also a critical variable. The model will incorporate this by multiplying the time taken to traverse each vector with the calculated probability of detection for that segment. This agrees with the intuitive notion that spending more time within a sensor's range increases the likelihood of being detected. Therefore, the speed at which each vector is traversed not only affects the overall time but also the detection risk.

Considering x_i is a checkpoint on the path. N represents the total number of checkpoints and δt is the time spent at each checkpoint. With p denoting probability of detection at that checkpoint. P being the cumulative probability score.

$$P = \sum_{i=1}^{N} p(x_i)\delta t$$

3.1.2 Speed and Its Implications

One of the unique features of this model is the incorporation of speed as an adjustable and independent variable for each vector connecting the checkpoints on the path. Traditional models incorporate speed often as a constant or a function of distance. However, in our model each vector will have an associated speed that can be optimised to meet various objectives and constraints.

The speed for each vector will be dynamically adjusted to ensure that the entire path is traversed within the given time constraint. This feature works in conjunction with other elements of the model, adding another layer of complexity and realism. Where S is the

$$S = \sum_{i=1}^{N} \frac{|v_i|}{\delta t}$$

Gradient Resistance and Speed
Additionally, a gradient resistance factor will be implemented. This factor will create a resistance based on the height elevation of a specific route. The speed of a specific vector will directly be affected by the "elevation disparity". For instance, climbing a steep hill would naturally slow down the subject, while descending could allow for increased speed. This adds another layer of complexity and realism to the model, making it more adaptable to real-world terrains and scenarios.

With S being the cumulative speed score and v_i denoting the velocity at a given checkpoint. Grad_i is the gradient of the vector v_i.

$$S = \sum_{i=1}^{N} \frac{|v_i|^{\mathrm{EXP}}}{\delta t}$$

$$\mathrm{EXP} = \lambda_3 + \frac{2}{\pi} \arctan(\mathrm{Grad}_i)$$

3.1.3 Time Constraint

In addition to the other parameters, a maximum time threshold will be introduced into the model. This serves as an upper limit on the traversal time for the agent on the path from the start to the end point. The time constraint ensures that the path not only minimises detection and adheres to the landscape but also is feasible within the given time frame. Where T is cumulative time score.

$$T = \sum_{i=1}^{N} \delta t$$

3.2 Challenges and Complexities

In a discrete formulation of the model, the solution space is vast. Choosing between a multitude of possible paths and scenarios is in itself a task. The complexity of finding an optimal solution is exacerbated by various additional considerations, making the efficiency of the algorithm a critical concern.

3.2.1 Efficiency Challenges

- **Number of Checkpoints:** The efficiency of the algorithm is inversely proportional to the number of checkpoints. More checkpoints mean more variables to optimize, increasing computational time.
- **Number of Sensors:** A higher number of sensors increases the complexity of calculating detection probabilities, thereby lowering efficiency.
- **Complexity of Terrain:** Varied terrain features add another layer of computational complexity, especially when considering gradient resistance.
- **Threshold Time:** Whenever the threshold time is exceeded, the path and speeds of vectors need to be readjusted, adding to the computational burden.
- **LoS Resistance:** LoS resistance calculation severly affects the computational time, from the intense calculations required. Reducing the λ intervals, leads to less LoS calculations which increases efficiency.

3.2.2 Higher Dimensions

Extending the model to higher dimensions, such as 2D or 3D, increases computational complexity. Factors like gradient resistance and line-of-sight from sensors add layers of complexity that must be efficiently managed.

3.2.3 Initial Path Considerations

The choice of the initial path could significantly affect the subsequent iterations of the algorithm. An inefficient starting point could lead the algorithm into local optima, affecting the quality of the final path.

3.2.4 Visualisation Challenges

Visualizing the model, especially in higher dimensions or with a large number of variables, poses its own set of challenges. Effective plots and visual aids are essential for interpreting the results and for debugging the algorithm.

3.3 Summary

In summary, the model aims to provide a comprehensive, realistic, and flexible solution to the problem of escaping sensors, adaptable to a wide range of scenarios and conditions.

4 Simulated Annealing

4.1 Introduction to Simulated Annealing

4.1.1 Concept of Simulated Annealing as an Optimisation Algorithm

Simulated Annealing (SA) is a probabilistic technique used for finding an approximate solution to an optimisation problem. Originating from the field of statistical mechanics, this algorithm is designed to explore the solution space of a given problem in a structured yet randomised manner. SA allows for a more flexible exploration than deterministic algorithms and avoids being stuck in local minima. It starts with an initial solution and iteratively refines it by making small, randomised changes. The acceptance of new solutions, even if they are worse than the current one, is governed by a probability function that decreases over time. This unique feature enables the algorithm to escape local minima and potentially discover a global minimum.

4.1.2 Analogy to Annealing in Metallurgy

The term ”simulated annealing” is inspired by the annealing process in metallurgy. Annealing is a heat treatment process where a material is heated to a high temperature and then gradually cooled down. The aim is to increase the size of the crystals in the material to eliminate defects. Thereby minimising internal stresses and most importantly reducing the system’s overall energy. Similarly, in simulated annealing, the ”temperature” of the system is metaphorically high at the beginning, allowing for a wide exploration of the solution space. As the ”temperature” decreases, the algorithm becomes more selective. This mimics the slow cooling process in annealing, which helps to find a state with minimised energy. In computational terms, an optimal solution.

4.1.3 Objective of Using Simulated Annealing in the Context of Escaping the Sensors

In the context of our problem the objective is to find the most efficient path that minimises the probability of detection while adhering to various constraints such as terrain, time, and speed.

The solution space for this problem is vast and filled with local minima that a deterministic algorithm might get stuck in. Simulated annealing allows us to incorporate various objectives and constraints into a single optimization model while escaping local minima. By doing so, we aim to find a path that is not just mathematically optimal but also practically feasible and adaptable to real-world scenarios.

4.2 Why Simulated Annealing Over Other Algorithms

4.2.1 Comparison with Other Optimisation Algorithms

Simulated Annealing (SA) is often compared with other optimisation algorithms like Steepest Gradient Descent, Genetic Algorithms, and Particle Swarm Optimization, among others. Each of these algorithms has its own merits and demerits, but they also have specific use-cases where they excel.

- **Steepest Gradient Descent**: This is a first-order optimisation algorithm that moves in the direction of the steepest decrease of the objective function. While it's computationally less expensive and easy to implement it can easily get stuck in local minima for non-convex functions.
- **Genetic Algorithms**: Inspired by the process of natural selection and using techniques such as mutation, crossover, and selection to find approximate solutions to optimisation problems. While they are good at exploring a large solution space, they often require a large population size and many generations to converge. Making them computationally expensive and too inefficeint for our use.

4.2.2 Advantages of Simulated Annealing

Simulated Annealing offers several advantages that make it suitable for a wide range of optimisation problems:

1. **Escaping Local Minima**: One of the most significant advantages is its ability to escape local minima. This feature is crucial for problems with complex, non-convex solution spaces.
2. **Flexibility**: SA is highly adaptable and can be tailored to fit various types of objective functions and constraints. This makes it a versatile tool for optimisation.
3. **Efficiency**: While it is a probabilistic algorithm, SA often requires fewer evaluations of the objective function.
4. **Ease of Implementation**: SA is relatively easy to implement and doesn't require the complex data structures that some other algorithms do.
5. **Tunability**: The algorithm's behavior can be easily adjusted by tuning parameters like the cooling schedule, allowing for a balance between exploration and exploitation of the solution space.

4.3 Detailed Overview of Simulated Annealing

4.3.1 Algorithm Steps

The algorithm starts with an initial solution and iteratively moves to a neighboring solution in the solution space. The acceptance of the new solution depends on an objective function and a temperature parameter. The steps of the algorithm are as follows:

1. Initialize $T_{\max}$, the maximum temperature.
2. Initialize $T_{\min}$, the minimum temperature.
3. Initialize s, the current solution.
4. While $T > T_{\min}$:
    (a) Generate a neighboring solution s'.
    (b) Calculate $\Delta E = f(s') - f(s)$, where f is the objective function.
    (c) If $\Delta E < 0$, accept s' as the new solution.
    (d) Else, accept s' with probability $P = e^{-\Delta E/T}$.
    (e) Update T using a cooling schedule, e.g., $T = \alpha T$, where $0 < \alpha < 1$.

4.3.2 Parameters

- $T_{\max}$ and $T_{\min}$: Control the starting and ending temperature.
- α: Cooling rate, which controls how fast the temperature decreases.

4.4 The SimAnneal Library

The SimAnneal library is a Python package implementation of the Simulated Annealing algorithm [7]. It provides a base class, called `Annealer`, which encapsulates the core logic of the algorithm, allowing users to focus on defining the problem-specific aspects such as the state transition and energy calculation.

4.4.1 Structure of an Annealer Subclass

The subclass should implement at least two methods: `move` and `energy`.

- **The move method:** This method defines how to move from one state to a neighbouring state in the solution space. It should modify the current state to a new state.

```
def move(self):
    # Implement logic to move to a neighboring state
```

- **The energy method:** This method calculates the energy (or cost) associated with a given state. Lower energy states are more desirable.

```
def energy(self):
    # Calculate and return the energy of the current state
```

4.5 Utilizing Simulated Annealing in the Algorithm

4.5.1 State Representation

In our specific problem, the state is represented as a sequence of coordinates that define a path from the starting point to the end point while avoiding sensors. Each coordinate in the sequence represents a checkpoint, and vectors connect these checkpoints to form the complete path. The state also includes additional variables such as the speed associated with each vector and the time taken to traverse each segment of the path.

4.5.2 The Move Method

The `move` method is implemented to generate a new state by perturbing the current state. This will involve altering a coordinate in the path, changing the speed associated with a particular set of vectors or adjusting the time between checkpoints. The objective is to explore the solution space effectively to find the optimal path.

4.5.3 The Energy Method

The `energy` method calculates the cost function for a given state. The cost function is designed to evaluate the quality of a path based on several criteria:

- The probability of detection by sensors, which is influenced by the path's proximity to sensors and the time spent near them.
- The total speed of the agent traversing the path
- The total time taken to traverse the path, which should be within a specified maximum time threshold.

The `energy` method combines these factors into a single numerical value that represents the "energy" or "cost" of the state.

4.6 The Cost Function

4.6.1 Definition in the Context of Optimisation

In optimisation problems, the cost function serves as a measure of the "quality" of different solutions. It quantifies how far a given solution is from the optimum, with the objective being to minimise this value. In the context of our problem, the cost function evaluates the effectiveness of a path based on multiple criteria such as detection probability, speed of the agent, and time.

4.6.2 Specific Cost Function in the Algorithm

The specific cost function $C(s)$ used in our algorithm is a weighted sum of several terms:

$$C(s) = \lambda_0 \cdot P(s) + \lambda_1 \cdot S(s) + \lambda_2 \cdot T(s)$$

Where:

- $C(s)$ is the total cost for state s.
- $P(s)$ is the probability of detection by sensors.
- $S(s)$ is the total speed of the agent.
- $T(s)$ is the total time taken to traverse the path.
- $\lambda_0, \lambda_1, \lambda_2$ are the weights for each term.

4.6.3 Adjusting Parameters for Different Scenarios

The weights $\lambda_0, \lambda_1, \lambda_2$ can be adjusted to prioritise different aspects of the problem depending on the scenario. For example, in a rescue mission, minimising time $T(s)$ might be more critical, so λ_2 would be set higher. Similarly, in a stealth mission, minimising detection probability $P(s)$ would be crucial, so λ_0 would be higher.

4.7 Changing Parameters in Simulated Annealing

4.7.1 Significance of the Number of Steps

The number of steps in the annealing schedule is a crucial parameter that affects the quality of the solution and the computational time. A higher number of steps allows the algorithm more opportunities to explore the solution space and potentially find a better minimum. However, this comes at the cost of increased computational time.

4.7.2 Role of Initial and Final Temperatures (Tmax and Tmin)

The initial temperature (T_{max}) and final temperature (T_{min}) are pivotal in controlling the behavior of the algorithm. T_{max} should be set high enough to allow the algorithm to explore a broad range of solutions initially, while T_{min} should be low enough to fine-tune the solution as the algorithm progresses.

4.7.3 Temperature Depreciation Function

The temperature depreciation function dictates how the temperature decreases over time or steps. This function plays a significant role in determining how fast the algorithm converges to a solution. Common depreciation functions include exponential decay and linear decay. The choice of function can affect the algorithm's ability to escape local minima and the overall time to find a solution.

4.8 Conclusion

In this section, we have delved into the intricacies of the simulated annealing algorithm. We have explored its theoretical foundation, advantages, and implementation details. We have also discussed the critical parameters that influence the algorithm's performance and how they can be fine-tuned for specific optimisation problems.

By leveraging the power and flexibility of simulated annealing, we aim to solve our complex optimization problem efficiently and effectively, thereby achieving our goal of finding the most optimal path under various constraints and considerations.

5 2D Model

5.1 Introduction to the 2D Model

The 2D model serves as the cornerstone of this entire project, laying the foundational framework upon which the more complex 3D model is built. It is in this 2D environment that the core algorithms and functions are first implemented and tested. Thus providing a simplified yet robust setting for initial experimentation and optimisation.

By reducing the problem to two dimensions, we can focus on the essential mechanics of our problem. This includes sensor detection, velocity calculations, and path optimisation without the added complexity of a third dimension. This allows for quicker iterations and a more straightforward debugging process, making it an invaluable tool for initial development.

In previous sections, we have laid down the theoretical and mathematical foundations that underpin this model. In this 2D model, all these components come together in a cohesive manner. Through this 2D model, we aim to validate our algorithms and set the stage for the more intricate 3D model that follows.

5.2 The Components of the Simulated Annealer

The simulated annealer serves as the heart of the 2D model, orchestrating the various components to find an optimal path through the sensor-laden landscape. Given its central role, it is crucial to understand the components that feed into it. These components are designed to quantify the quality of a particular path.

5.2.1 Probability of Detection Functions

The probability of detection is a critical metric in our model, as it quantifies the risk associated with a given path. To calculate this, we employ a series of functions that work in tandem.

Dist2Sens(Sens, Point)

```
# Calculate the Euclidean distance between each sensor and the point
def Dist2Sens(Sens, Point):
    distances = np.linalg.norm(Sens - Point, axis=1)
    return distances
```

- **Purpose**: This function calculates the Euclidean distance between a sensor (`Sens`) and a point (`Point`) in the 2D space.
- **How it Works**: The function takes the coordinates of a sensor and a point as input and applies the standard formula for Euclidean distance to calculate the distance between them.
- **Role in Algorithm**: The distance calculated by this function is then used to determine the probability of detection by that particular sensor, feeding into the `PropSens` function.

PropSens(dist, sig = 1)

```
# Apply sigmoid function to model sensor's detection probability
def PropSens(dist, sig = 1):
    p = 1/(1+((dist/sig)**2))
    return p
```

- **Purpose**: This function models the probability of detection based on the distance (`dist`) from a sensor.
- **How it Works**: The function uses a sigmoid function to model the probability of detection. The `sig` parameter controls the steepness of the sigmoid curve, allowing for adjustments based on different sensor characteristics. The larger `sig` is, the higher the probability of detection.
- **Role in Algorithm**: The output of this function gives the likelihood of being detected at a particular distance from a sensor, which is crucial for evaluating the risk associated with a given path.

ProbDetect(Sensors, Point)

```
def ProbDetect(Sensors, Point):
    # Get distances from all sensors to the point
    distances = Dist2Sens(Sensors, Point)
    # Select the two closest sensors
    twoSens = np.argsort(distances)[:2]
    # Calculate the final probability of detection
    p1 = PropSens(distances[twoSens[0]])
    p2 = PropSens(distances[twoSens[1]])
    P = p1*p2
    return P
```

- **Purpose**: This function integrates the outputs of `Dist2Sens` and `PropSens` to calculate the overall probability of detection at a given point.
- **How it Works**: The function selects the two closest sensors to the point in question and calculates the probability of detection based on the distances to these sensors. It then multiplies these probabilities to get the final probability of detection for that point.
- **Role in Algorithm**: This is a key function in the simulated annealing algorithm, as it provides a composite measure of risk for each point along the path. This risk assessment is then used to evaluate and optimise the path.

5.2.2 Velocity Functions

The calculated velocities are crucial for determining the speed of each segment of the path, which in turn affects the overall risk assessment and optimisation process in the simulated annealing algorithm.

Points2Velocity(Points)

```
def Points2Velocity(Points):
    V = []  # Initialize an empty list to store velocity vectors
    numVelocity = len(Points) - 1  # Number of velocity vectors to be calculated
    for i in range(numVelocity):
        Start = Points[i]  # Starting point of the segment
        End = Points[i + 1]  # Ending point of the segment
        direction = End - Start  # Calculate the direction vector
        r = np.linalg.norm(direction)  # Calculate the magnitude of the vector
        theta = np.arctan(direction[1] / direction[0])  # Calculate the angle of the vector
        V.append((r, theta))  # Append the magnitude and angle as a tuple to the list V
    return V  # Return the list of velocity vectors
```

- **Purpose**: This function calculates the velocity vectors between consecutive points along the path. Each velocity vector is represented by its magnitude (r) and direction (θ).
- **How it Works**: The function iterates through the list of points that make up the path. For each pair of consecutive points, it calculates the direction vector and then computes its magnitude and angle. These two values are then stored as a tuple in the list V, which is returned as the output.
- **Importance of Velocity**: Calculating the velocity of an agent on the path is essential for several reasons. First, it allows for a more realistic representation of movement, taking into account the time factor. Second, varying the velocity can lead to different levels of detection risk and can be adjusted to meet time constraints. There are different methods to handle velocity changes, such as constant acceleration or predefined speed limits, which can be incorporated depending on the specific requirements of the scenario.

5.2.3 Prerequisites

The prerequisites are essential for setting up the initial conditions for the simulated annealing algorithm. They help in defining the starting point and the time constraints for the optimisation process.

StartSolPoints(Start, End)

```
def StartSolPoints(Start, End):
    points = []  # Initialize an empty list to store points
    direc = End - Start  # Calculate the direction vector
    dist = np.linalg.norm(direc)  # Calculate the distance between Start and End
    numPoints = 20 + 1  # Number of points to be generated
    stepSize = dist / numPoints  # Calculate the step size
    for Lambda in range(numPoints + 1):
        point = Start + stepSize * Lambda * (1 / dist) * direc  # Generate each point
        points.append(point)  # Append the point to the list
    return points  # Return the list of points
```

- **Purpose**: This function generates an initial set of points between the starting and ending positions. These points serve as the initial solution for the simulated annealing algorithm.
- **How it Works**: The function calculates the direction vector between the starting and ending points and divides it into equal segments. It then generates points along this direction at equal intervals.
- **Role in Algorithm**: This function provides the initial state for the simulated annealing algorithm, which is crucial for kick-starting the optimization process.

TotalT_ThreshCalc(Start, End, Min_Speed = 1.5)

```
def TotalT_ThreshCalc(Start, End, Min_Speed = 1.5):
    # Calculate the distance between Start and End
    Dist = np.linalg.norm(End - Start)
    Time = Dist / Min_Speed  # Calculate the total time threshold
    return Time  # Return the total time threshold
```

- **Purpose**: This function calculates the total time threshold based on the minimum speed and the distance between the starting and ending points.
- **How it Works**: The function takes the starting and ending points and the minimum speed as inputs. It calculates the distance between the points and divides it by the minimum speed to get the total time threshold.
- **Role in Algorithm**: This function is essential for ensuring that the simulated annealing algorithm adheres to time constraints, thereby making the solution more realistic and applicable.

5.3 Simulated Annealing: The Heart of the Algorithm

The simulated annealing algorithm is the cornerstone of the 2D model. It is responsible for optimising the path by iteratively exploring the solution space to find a path that minimises the risk of detection, adheres to time constraints, and optimises the agent's speed.

Step 1. random_state(self)

```
def random_state(self):
    # Initialize the starting and ending points
    points = StartSolPoints(start, end)
    # Calculate the velocity vectors between consecutive points
    vectors = Points2Velocity(points)
    # Calculate the time assigned between each point
    T = TotalT_ThreshCalc(start, end, Min_Speed = 2.5)/len(vectors)
    return vectors, points, T
```

- **Purpose**: This function initialises the starting state for the simulated annealing algorithm. Unlike typical random initialisations, this function starts with a strong path, which is a straight line between the start and end points.
- **How it Works**: The function uses `StartSolPoints` to create an initial set of points that make up the path. It then calculates the velocity vectors between these points using `Points2Velocity`. Finally, it calculates the time T assigned between each point using `TotalT_ThreshCalc`.
- **Initial State Components**: The function returns three components: `vectors`, `points`, and T. Here, `vectors` are the velocity vectors between consecutive points, `points` are the coordinates of the points along the path, and T is the time assigned between each checkpoint. This time is the same for every section, and the velocities are determined by the different distances between points.

Step 2. energy(self)

```
def energy(self):
    # Retrieve the current state components: Vectors, Points, and T
    Vectors = self.state[0]
    Points = self.state[1]
    T = self.state[2]

    # Initialize the probability of detection (P)
    P = 0
    for i in range(len(Points)):
        P += ProbDetect(sensors, Points[i])

    # Initialize the speed (S)
    S = 0
    for i in range(len(Vectors)):
        S += (Vectors[i][0])**2/T

    # Calculate the total time (TotalT)
    TotalT = T * len(Vectors)

    # Calculate the energy (E) based on the cost function
    E = float(lambda0 * P * T + lambda1 * S + lambda2 * TotalT)

    # Append the energy value for later analysis
    iter_E.append(E)

    return E
```

- **Purpose**: This function calculates the energy or 'cost' of a given state, which serves as a measure of its optimality.
- **How it Works**: The function retrieves the current state components (`Vectors`, `Points`, and T) and calculates the probability of detection (P), speed (S), and total time (TotalT).
- **Cost Function**: The energy E is calculated based on a weighted sum of P, S, and TotalT, where $\lambda 0$, $\lambda 1$, and $\lambda 2$ are the coefficients for the probability of detection, speed, and time, respectively.
- **Storing Energy Values**: The function appends the calculated energy value to the list `iter_E` for later analysis.

Step 3. move(self)

```
def move(self):
    # Retrieve the current state components: Vectors, Points, and T
    Vectors = self.state[0]
    Points = self.state[1]
    T = self.state[2]

    # Randomly select a point to perturb
    i = random.randint(1, len(Points) - 2)

    # Initialize the current total time to exceed the threshold
    TotalT_Curr = TotalT_Thresh + 10

    while TotalT_Curr > TotalT_Thresh:
        # Perturb the direction and magnitude of the selected point
        adjustTheta = 1
        angle = np.random.uniform(-np.pi * adjustTheta, np.pi * adjustTheta)
        adjustR = 0.1
        distR = np.random.uniform(0, adjustR)

        # Update the point
        Points[i] = Points[i] + np.array([distR * np.cos(angle), distR * np.sin(angle)])

        # Update the velocity vectors
        Vectors = Points2Velocity(Points)

        # Adjust the time
        adjustT = 0.1
        magnitude_factorT = np.random.uniform(1 - adjustT, 1 + adjustT)
        T = T * magnitude_factorT

        # Calculate the current total time
        TotalT_Curr = T * len(Vectors)

    # Store the new state components for later analysis
    iter_Vectors.append(Vectors)
    iter_Points.append(Points)
    iter_T.append(T)
    # Update the state
    self.state = Vectors, Points, T
```

- **Purpose**: The primary objective of this function is to explore the solution space by generating new candidate solutions. It does this by perturbing the direction and magnitude of a randomly selected point along the path.
- **How it Works**: The function starts by retrieving the current state components: Vectors, Points, and T. A random point is then selected for perturbation. The function employs a while loop to ensure that the total time to traverse the path does not exceed a predefined threshold. Within this loop, the function perturbs the direction and magnitude of the selected point using random adjustments - updating Points. The velocity vectors and time T are then updated accordingly.
- **Perturbation Strategies**: There are three common approaches to perturbing a path: 1) perturbing one point per iteration, 2) perturbing several points, and 3) perturbing all points at once. In this implementation, we chose to perturb one point per iteration. This approach offers a fine-grained control over the solution space, allowing for more nuanced exploration. It is also computationally less expensive compared to perturbing multiple points.
- **Time Constraint**: The time threshold TotalT_Thresh serves as a constraint to ensure that the total time to traverse the path is within acceptable limits. If the total time exceeds this threshold, the path is rejected, and the process repeats.
- **Storing State Components**: For later analysis or visualisation, the function stores the new state components (Vectors, Points, and T) in separate lists.

5.4 Visualisation of the 2D Model

Visualisation is an indispensable tool in the realm of pathfinding algorithms. While numerical outcomes are essential for quantitative analysis, a visual representation provides an intuitive understanding of the algorithm's performance. This is particularly beneficial for individuals who may not be familiar with Python or have no experience in pathfinding.

5.4.1 Importance of Good Path Visualisation

A well-visualised path should meet several criteria. It should deviate away from sensor locations to minimise detection risk, maintain a reasonable length to be practical, and exhibit higher speeds near sensors to further reduce the probability of detection. The path should also start and end at the designated points, providing a complete solution to the problem.

5.4.2 Plotting Function

The plotting function, `Plot`, serves multiple purposes. It not only plots the optimised path but also visualises the sensor locations and speed variations along the path. One of the key features of this function is the heatmap, which provides a color-coded representation of detection probabilities across the grid. This heatmap is generated using the `ProbDetect` function and offers a visual cue for areas with high detection risk.

```
# Start and End Points
start = np.array([1, 5])
end = np.array([7, 5])
```

```
# Sensors
sensors = np.array([[4, 6], [3, 4]])

# Initial Temperature and Lambda Coefficients
initial_temp = 25000
lambda0 = 11500
lambda1 = 10000
lambda2 = 1000

# Total Time Threshold
TotalT_Thresh = TotalT_ThreshCalc(start, end, Min_Speed = 0.01) + 5
```

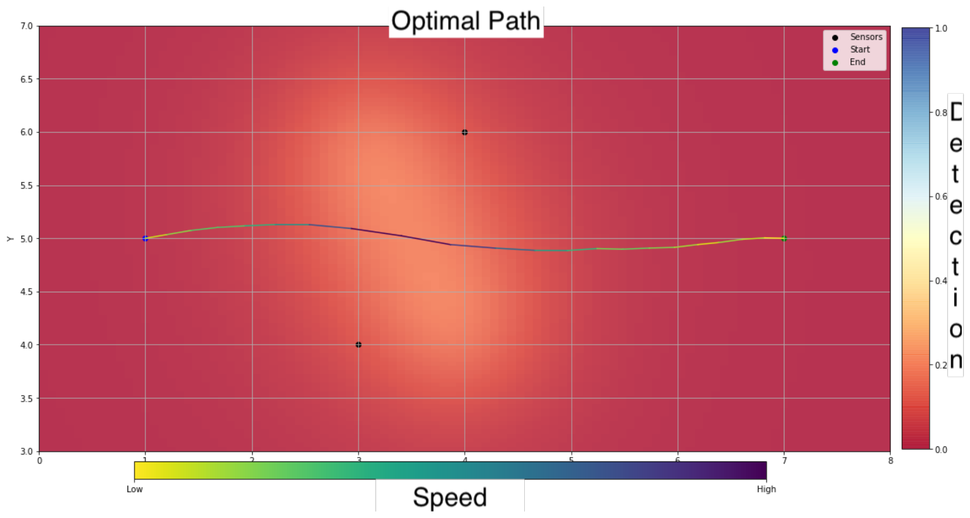

Figure 1: Optimal Path, generated by simulated annealer

5.4.3 Dynamic Visualisation

To further aid in understanding the algorithm's behavior over time, a video visualization was also implemented using the function `Visualize`. This video comprises different paths at various iterations, showcasing the algorithm's exploration and convergence behavior. The chaotic paths at the beginning represent the algorithm's exploration phase, which gradually converges to an optimal or near-optimal solution.

5.4.4 Energy Plot

Another insightful visualisation tool is the energy plot, generated by the function `PlotE`. This plot displays the energy values at each iteration. The initial high-energy fluctuations signify the algorithm's exploration phase, enabled by the high 'temperature'. As the algorithm progresses and the 'temperature' decreases, the energy values start to converge. Occasional spikes in energy values represent the algorithm's ability to escape local minima. A line of best fit, represented in red, shows an exponential decrease in energy over time, confirming the algorithm's convergence to a global minimum or a near-optimal solution.

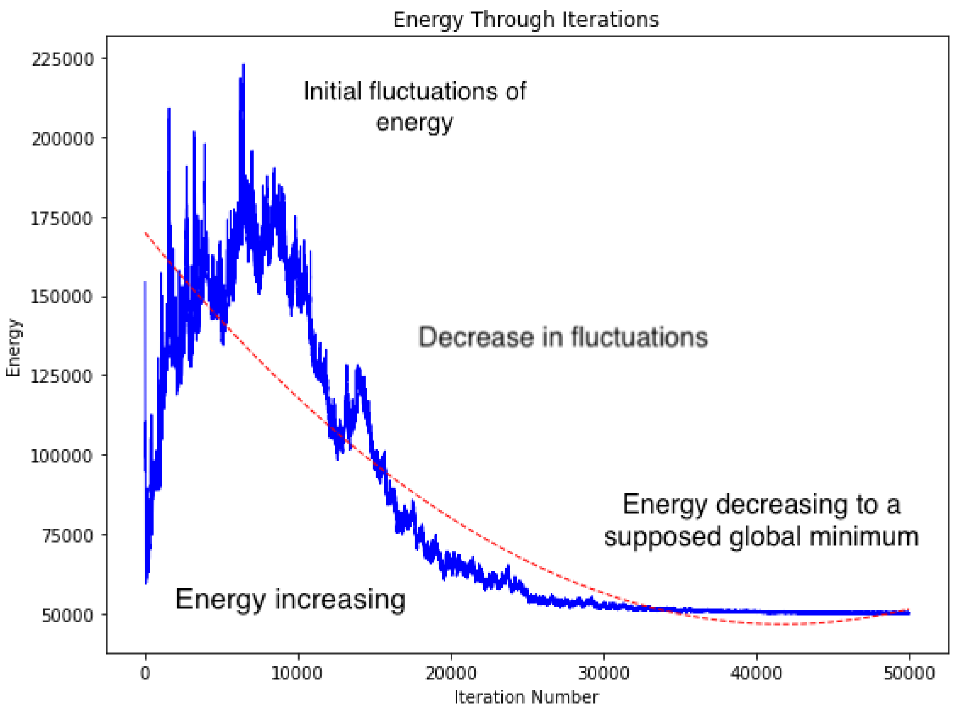

Figure 2: Energy values over iterations

5.4.5 Further examples of the 2D model

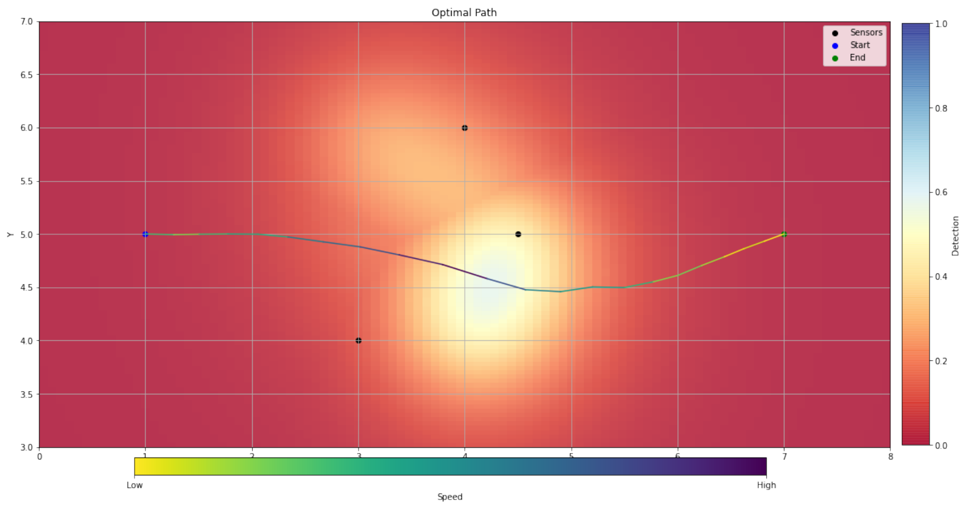

Figure 3: Three randomly placed sensors

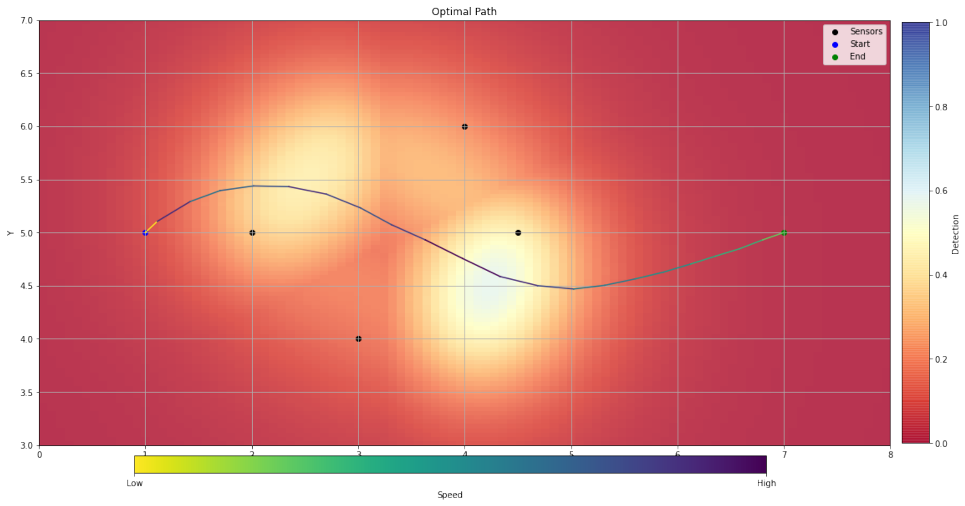

Figure 4: Four randomly placed sensors

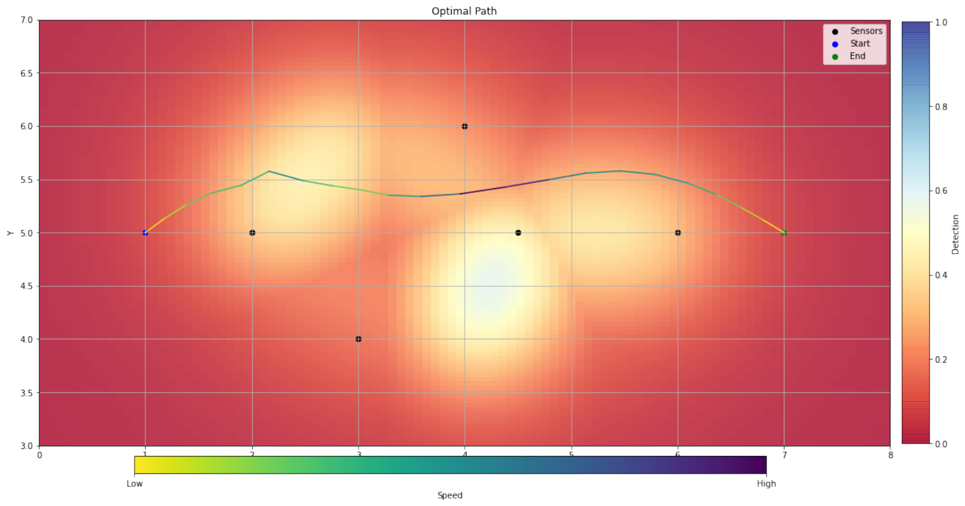

Figure 5: Five randomly placed sensors

5.5 Conclusion for the 2D Model

The 2D model serves as the foundational layer for the entire project, providing a simplified yet robust framework for path optimisation. It integrates various components, such as probability of detection, velocity of the agent, and time constraints, into a cohesive model. The simulated annealing algorithm lies at the heart of this model, offering a powerful mechanism for exploring the solution space and finding optimal paths under given constraints.

As we can see from the examples of the 2D models given. The paths are clearly avoiding the sensors and distributing the speeds along the vectors logically. The model adapts well to an increase in sensors and behaves as expected.

In summary, the 2D model successfully integrates various mathematical and algorithmic components to provide a comprehensive solution for path optimisation. Its modular design allows for easy adaptability and serves as a strong foundation for extending the model to more complex scenarios, including the forthcoming 3D model.

6 3D Model with Topology

The 3D model serves as an extension of the foundational 2D model, incorporating an additional dimension to represent height or elevation. This transition from a 2D to a 3D representation is significant for applications that require path finding over complex terrains. The key difference between the 2D and 3D models lies in the introduction of a surface, which adds a layer of complexity and realism to the problem.

The paper [12], researches ”Intelligent 3D Path Finding”. The researches seek to ”improve and optimise the A^* algorithm in two-dimensional path finding, and this way can meet the requirements of calculation in three-dimensions”. This is the same direction we will be taking. The paper serves to validate our method with their similar techniques used.

6.1 The Height Function

The `Height(Position)` function is the cornerstone of the 3D model, setting it apart from its 2D counterpart. This function takes a position defined in the 2D plane and assigns it a height, thereby transforming the plane into a 3D surface. The plot is also adjusted using the height function. Contour lines are drawn on all graphs to represent areas of elevation.

6.1.1 Function Details

The code for the `Height(Position)` function is as follows:

```
def Height(Position):
    x = Position[0]
    y = Position[1]
    # Calculate the height based on the given x and y coordinates
    # Example height function
    h = np.exp(-((x-2.5)**2+(y-2)**2)*2)*2+np.exp(-((x+1.5)**2+(y+1)**2)*2)*2
    return [x, y, h]
```

The function uses a mathematical expression to calculate the height h based on the x and y coordinates. It is crucial for this function to be continuous to ensure that the path finding algorithm can operate effectively over the surface.

6.1.2 Real-world Applications

The height function can be adapted to represent real-world terrains. For instance, one could use topographical data to create a height function that mimics a specific landscape. Additionally, interpolation techniques can be applied to gridded landscapes to generate a continuous function, making the model adaptable to real-world scenarios.

6.2 Gradient Resistance

In a 3D model, the gradient of the surface plays a crucial role in determining the resistance encountered while traversing the path. For instance, moving uphill is generally more ”expensive” in terms of energy cost compared to moving downhill. This section discusses how the model accounts for gradient resistance.

6.2.1 Calculating Gradient and Velocity

The function `Points2VelocityNGrad` calculates both the velocity vectors and the gradients between each pair of points. The code for this function is as follows:

```
def Points2VelocityNGrad(Points):
    V = []  # Initialize an empty list to store velocity vectors
    Grads = []  # Initialize an empty list to store gradients
    numVelocity = len(Points) - 1  # Calculate the number of velocity vectors to be generat

    # Loop through each pair of points to calculate velocity vectors and gradients
    for i in range(numVelocity):
        # Get the 3D coordinates of the start and end points using the Height function
        Start = np.array(Height(Points[i]))
        End = np.array(Height(Points[i+1]))

        # Calculate the vector from the start point to the end point
        Vector = End - Start

        # Calculate the length of the vector in the xy-plane
        L = np.sqrt(Vector[0]**2 + Vector[1]**2)

        # Extract the change in height (Z component of the vector)
        Z = Vector[2]

        # Calculate the gradient as the change in height divided by the length in the xy-pl

        Grad = Z / L

        # Append the calculated vector and gradient to their respective lists
        V.append(Vector)
        Grads.append(Grad)

    # Return the lists of velocity vectors and gradients
    return V, Grads
```

In this function, the vectors are stored as normal vectors instead of in (r, θ) form for simplicity. The gradient is calculated as $\frac{Z}{L}$, where Z is the change in height and L is the length of the vector in the xy-plane.

6.2.2 Modifying the Move Function

The `move` function is slightly altered to calculate both vectors and gradients when new points are generated. The gradients are stored for later use in the energy calculation.

```
def move(self):
        # Previous code iterating through each point in the state and modifying
        its direction and magnitude

        Vectors, Grads = Points2VelocityNGrad(Points) <--

    iter_Vectors.append(Vectors)
    iter_Points.append(Points)
    iter_T.append(T)
    iter_Grads.append(Grads) <--

    self.state = Vectors, Points, T
```

6.2.3 Energy Calculation with Gradient Resistance

The energy function now incorporates the effect of gradient resistance, which is proportional to speed. The code for the `energy` function is as follows:

```
def energy(self):
    #Previous code

    S = 0
    for Vector in range(len(Vectors)):
        Grad = iter_Grads[-1][Vector]
        EXP = lambda3+(2/np.pi)*np.arctan(Grad)
        S += (np.linalg.norm(Vectors[Vector]))**EXP/T

    #energy function continuous...
```

In this function, the speed S now has an exponential component and is calculated as a function of the gradient.

$$S = \frac{|v|^{\text{EXP}}}{T}$$

$$\text{EXP} = \lambda_3 + \frac{2}{\pi}\arctan(\text{Grad})$$

The EXP term is introduced to account for the gradient resistance.

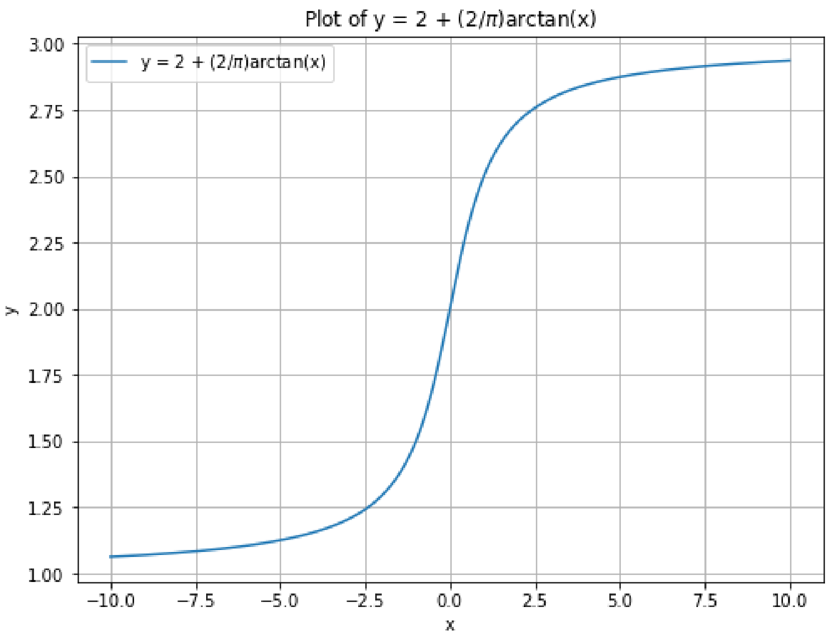

Figure 6: EXP function against gradient

The EXP term determines how much of a "boost" the value for speed (S) recieves. As the gradient increases the EXP term increases, which in turn increases the speed. The higher the speed the higher the cost. This ensures that the energy cost is appropriately adjusted based on the surface gradient.

6.3 Line of Sight Resistance

6.3.1 Importance in a 3D Environment

While we have already accounted for the fall-off in sensor detection probability with distance, it is crucial to consider the impact of the terrain in obstructing the sensor's line of sight. This leads us to the concept of *Line of Sight Resistance*, which quantifies the degree to which the terrain blocks or reduces the sensor's field of view, thereby affecting the probability of detection.

6.3.2 Code Explanation

The code for calculating the Line of Sight Resistance is as follows:

```
def LineOfSightResistance(Position, Sensor, interval = 0.1, strength = 0.2):
    Position = Height(Position)
    Sensor = Height(Sensor)
    dist = np.linalg.norm(np.array(Sensor) - np.array(Position))
    if dist == 0:
        Resistance = 0
        return Resistance
    Direc = (np.array(Sensor) - np.array(Position))/dist
    Lambda = 0
    CheckP = 0
    NLoSCheckP = 0
    while Lambda <= dist:
        CheckP += 1
        L = Position + Lambda*Direc
        HeightL = L[2]
        HeightS = Height([L[0],L[1]])[2]
        if HeightS > HeightL:
            NLoSCheckP += 1
        Lambda += interval
    NLoSPercentage = NLoSCheckP/CheckP
    Resistance = (-strength*NLoSPercentage)+1
    return Resistance**2
```

$$\text{Position} = \begin{pmatrix} x \\ y \end{pmatrix}, \quad \text{Sensor} = \begin{pmatrix} a \\ b \end{pmatrix}$$

The function first converts the 2D positions of the sensor and the object to 3D by adding height information using the `Height` function.

$$\begin{pmatrix} x \\ y \end{pmatrix} \to \begin{pmatrix} x \\ y \\ z \end{pmatrix}, \quad \begin{pmatrix} a \\ b \end{pmatrix} \to \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

It then calculates the Euclidean distance between the sensor and the object. If the distance is zero, the resistance is set to zero.

$$\text{dist} = \sqrt{(a-x)^2 + (b-y)^2 + (c-z)^2}$$

$$\text{if dist} = 0, \text{ then Resistance} = 0$$

The direction vector from the object to the sensor is calculated.

$$\text{Direc} = \frac{(a-x, b-y, c-z)}{\text{dist}}$$

A loop iterates through points at a specified interval along the line connecting the sensor and the object.

$$L = \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \lambda \times \text{Direc}$$

At an arbitrary point along the line say:

$$L = \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix}$$

$$Height(\alpha, \beta) = h$$

$$\text{if } h > \gamma \text{ then } NLoSCheckP+=1$$

If the height of the point on the line 'γ' is higher than the height of the surface 'h' the point is recorded as NLoS (no line of sight). This is to check if it is obstructed by the terrain. This process is repeated along all chosen points along the line.

$$\text{NLoSPercentage} = \frac{NLoSCheckP}{CheckP}$$
$$\text{Resistance} = (-\text{strength} \times \text{NLoSPercentage}) + 1$$

The NLoS Percentage is found and used to calculate the resistance.

The parameter `strength` controls the impact of the line of sight obstruction on the resistance. A higher value means that obstructions have a more significant impact, while a lower value means they have less impact.

6.4 Examples of the Final Model

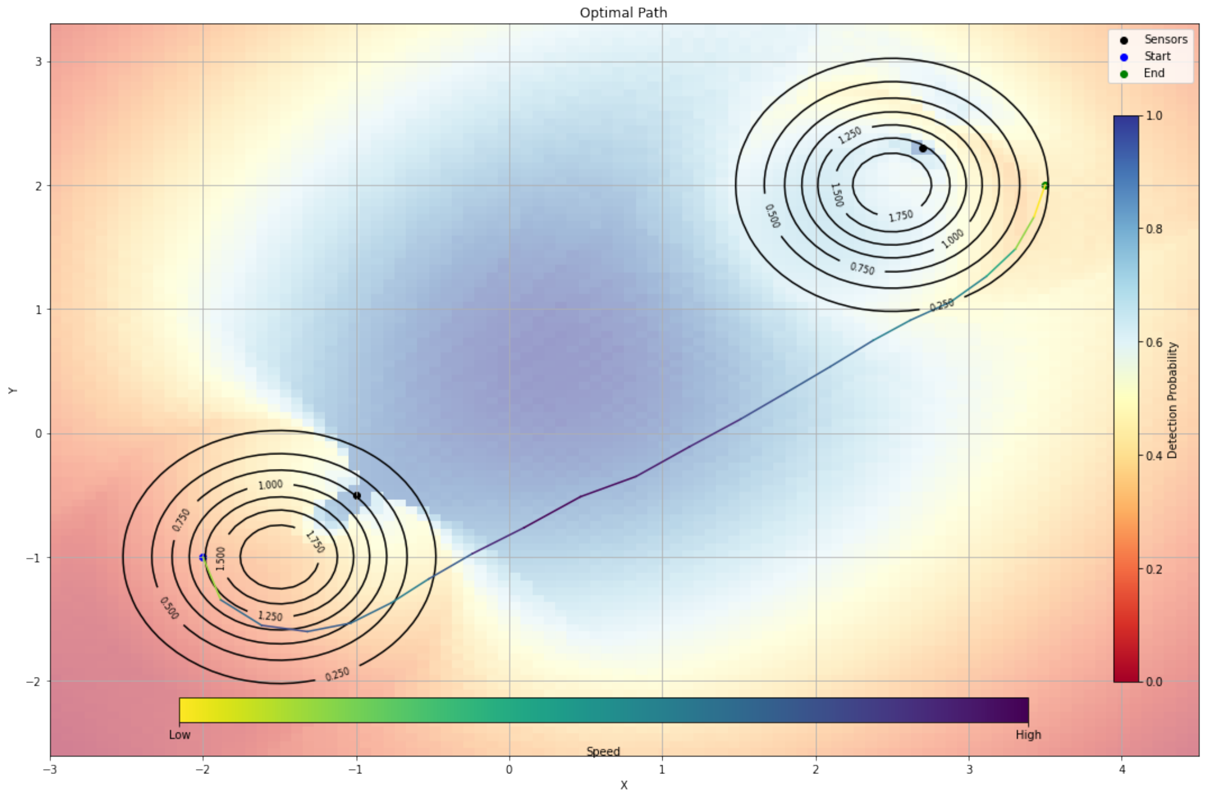

Figure 7: Final Model Original

Here we can explicitly see the differences between the 3D model and the 2D model. Assessing the heatmap, the blue areas representing high detection, are highly dependant on LoS not just proximity. The sensor at $(-1, -0.5)$ is placed on the side of a hill. Visualising the contour at this location, a large amount of the terrain in the direction of the hill will be obscured by the hill itself. However, the elevation of the sensor has allowed it LoS to much more terrain in the direction away from the hill. Hence why the detection is at its strongest in the middle of the map, and weak on the hill.

As a testament to the gradient resistance the path can be seen to avoid travelling along positive gradients. Observing the path on the hill at $(-1.5, -1)$, the path avoids the more direct route over the hill to favour going around, displaying the work of gradient resistance.

Further examples are shown below:

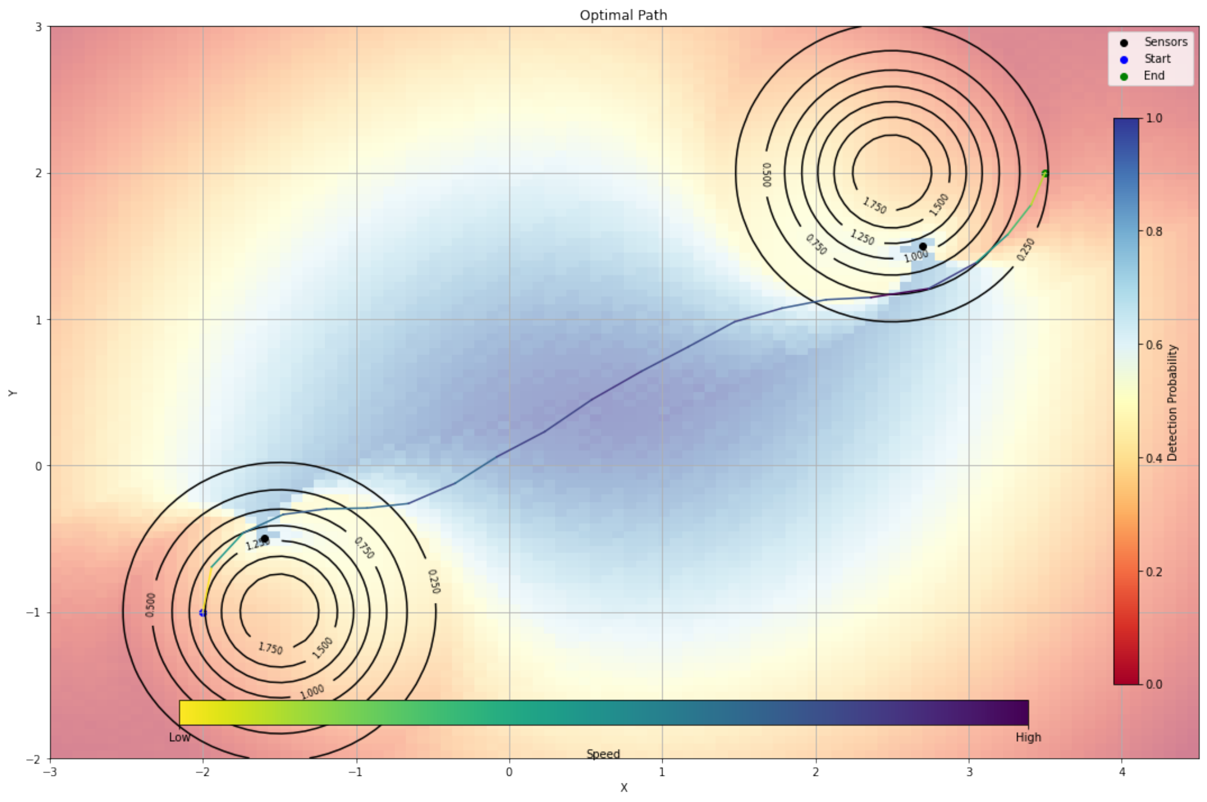

Figure 8: Final Model Config 2

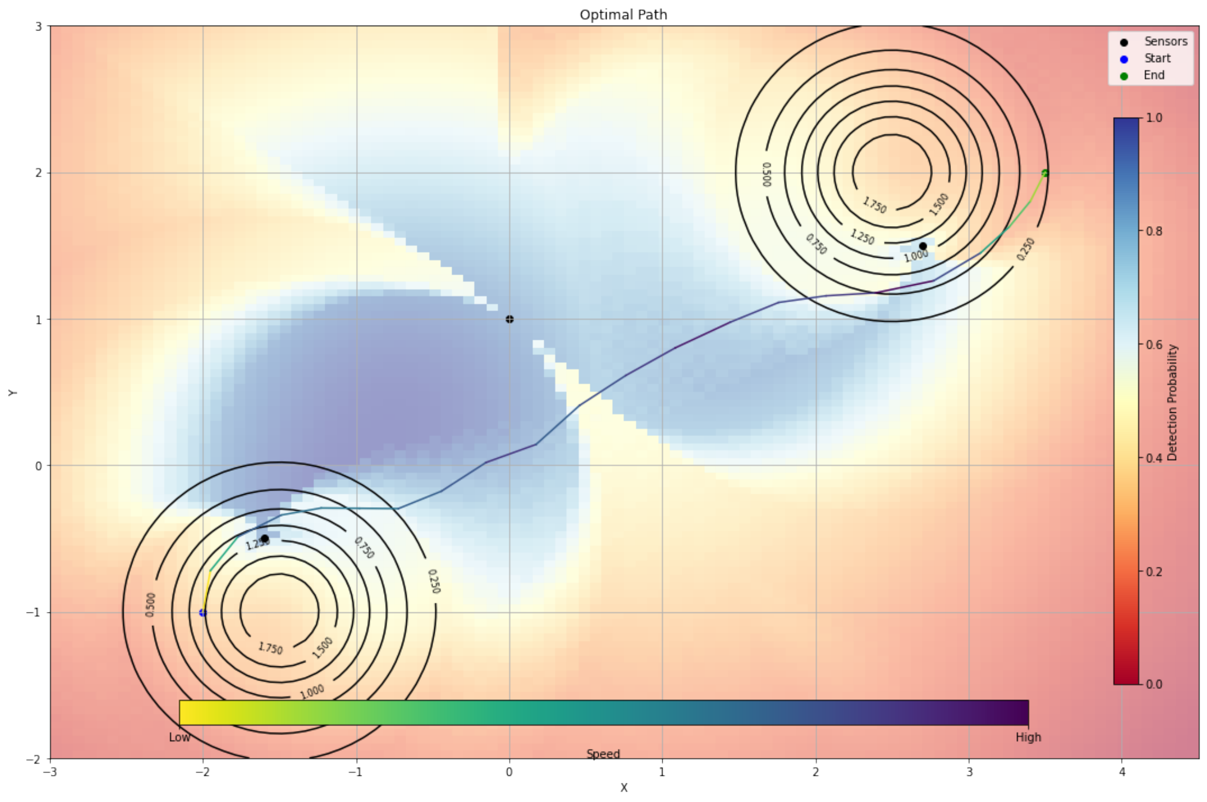

Figure 9: Final Model 3 Sensors

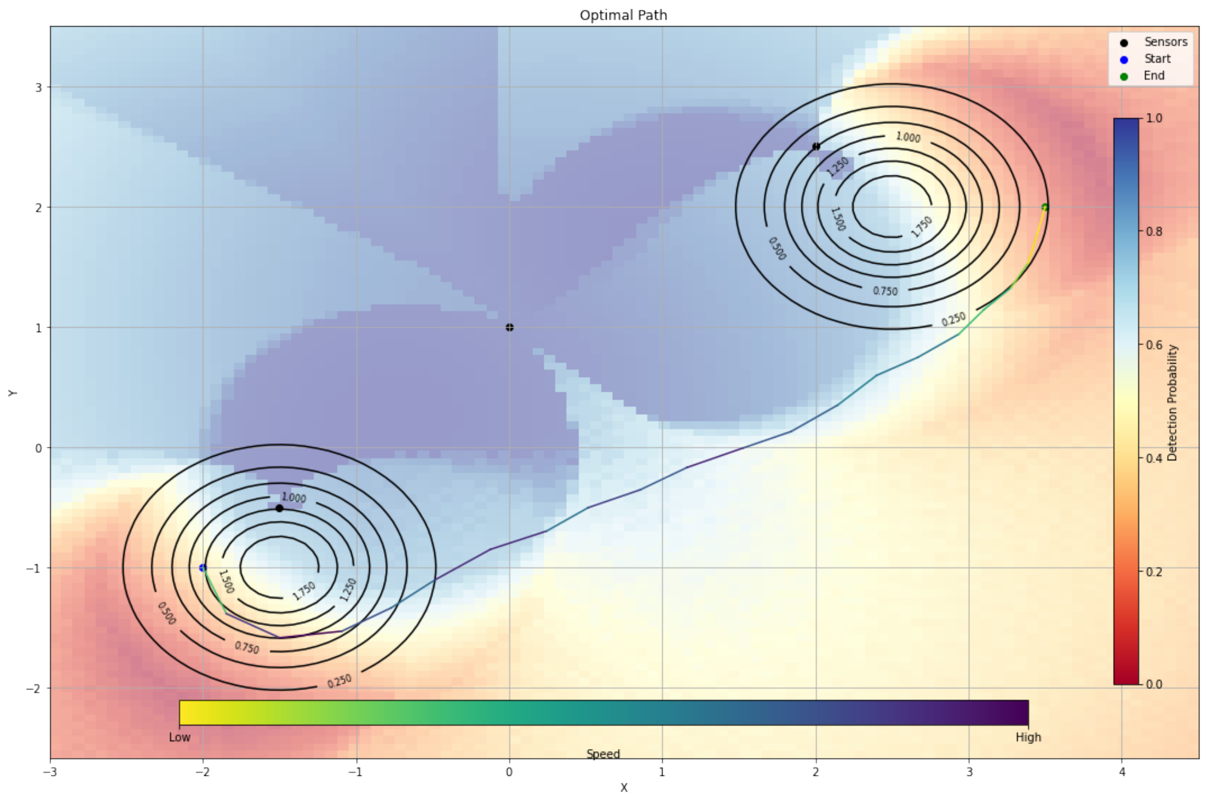

Figure 10: Final Model 3 Sensors Config 2

7 Analysis of the Final Model

The analysis of the final model is bifurcated into two primary sections. The first section delves into the parameters of the simulated annealing algorithm, specifically evaluating the roles and impacts of $T_{\max}$, $T_{\min}$, and the step values. The objective is to provide a general understanding of which parameters are most influential in determining the performance of the model.

The second section will extend this analysis to discuss the implications of these parameters on the model's overall performance, including metrics such as traversal time, detection probability, and energy cost.

It is important to note that the optimal parameter values are highly dependent on the specific scenario at hand. Therefore, while this analysis aims to offer a general guideline for parameter selection, the ultimate fine-tuning will require a trial-and-error approach tailored to the individual scenario. This discussion serves as a foundational step in that direction, offering data-driven insights into the general importance of each parameter.

The paper [14], provides an example of an excellent paper on analysing path finding algorithms. The paper provides a theoretical and practical part for analysis. While also including execution time and memory requirements as part of it. This will serve as inspiration moving forward with the analysis.

7.1 Simulated Annealer Parameter Tuning

7.1.1 Prerequisites

Before diving into the analysis, it is crucial to establish the prerequisites that set the stage for evaluating the simulated annealing parameters. The first step involves defining a terrain that has an obvious optimal solution. This is essential for consistently determining how close the final cost is to this optimal solution.

To achieve this, a terrain was designed with a significant valley, and both the start and end points were placed within this valley. The optimal solution in this case is a straight line connecting the start and end points, which minimizes distance, proximity to sensors, and gradient.

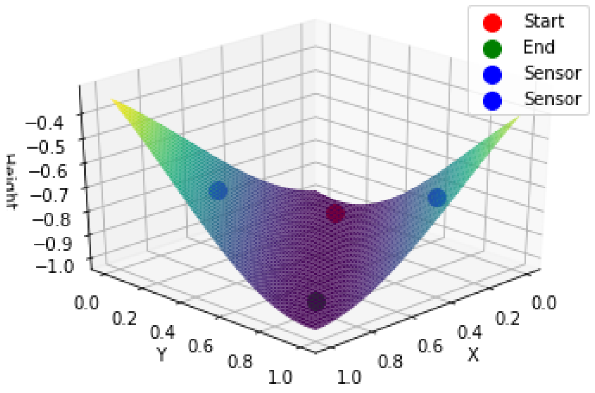

Figure 11: Terrain defined for analysis

However, the original final model started with a straight line between the two points, which is already the optimal solution. To avoid this, the starting solution was altered to consist of random points. The Python function `StartSolPoints` was used for this purpose:

```
def StartSolPoints(Start, End, noise_scale = 8.5):
    points = []
    direc = np.array(End) - np.array(Start)
    dist = np.linalg.norm(direc)
    numPoints = 20 + 1
    stepSize = dist / numPoints

    for Lambda in range(numPoints + 1):
        # Linear interpolation between start and end
        point_linear = Start + stepSize * Lambda * (1/dist) * direc

        # Add some random noise to each coordinate to make the point random
        noise = np.random.normal(0, noise_scale, size=len(Start))
        point_random = point_linear + noise

        points.append(point_random)

    points[0] = np.array(Start)
    points[-1] = np.array(End)

    return points
```

This approach differs from the initial straight-line solution and required fine-tuning the strength of the randomness to ensure a good starting point for the simulated annealing algorithm.

The quality of the final solution serves as a clear indicator of the effectiveness of the simulated annealing algorithm, making it a critical aspect of this analysis.

7.1.2 Experiment Configuration

The parameters chosen for testing are $T_{\max}$, $T_{\min}$, and the number of steps. These parameters were selected because they are the most influential in determining the annealing schedule, which in turn affects the quality of the final solution.

The values for $T_{\max}$ were set to $[10000.0, 15000, 25000.0, 50000.0]$, for $T_{\min}$ to $[0.1, 1, 2.5, 5.0]$, and for the number of steps to $[20000, 50000, 75000, 100000]$.

Each configuration was repeated 5 times to account for the stochastic nature of the algorithm and to provide a more reliable estimate of its performance.

Logging and Data Collection The results of each run were logged individually in JSON files. This approach was chosen to allow for the resumption of the experiment in case of interruptions or crashes. Each JSON file contains the parameters used, the final energy, the time taken, and the convergence time.

Performance Metrics The key metrics for evaluating the performance of the simulated annealer are the final cost, the time taken, and the convergence time. The final cost gives an indication of the quality of the solution, the time taken provides insights into the efficiency of the algorithm, and the convergence time helps in understanding how quickly the algorithm stabilises.

Defining Convergence Time Convergence time is defined as the time step at which the rate of change of energy falls below a certain threshold. The Python function `find_convergence_time` was used to calculate this:

```
def find_convergence_time(iter_E, threshold=0.001):
    """
    Find the time step where the energy (or cost) becomes stable.

    Parameters:
        iter_E (list): List of energy (or cost) values at each time step.
        threshold (float): The threshold for the rate of change of energy.

    Returns:
        int: The time step where the energy becomes stable.
            Returns -1 if the energy never stabilizes within the given threshold.
    """
    for i in range(1, len(iter_E) - 1):
        # Calculate the rate of change of energy
        rate_of_change = abs(iter_E[i] - iter_E[i - 1]) / iter_E[i]

        # Check if the rate of change is below the threshold
        if rate_of_change < threshold:
            return i
    return -1  # Return -1 if never converges within the given threshold
```

This definition was chosen because it provides a quantitative measure of when the algorithm has effectively stopped improving, which is crucial for understanding its performance.

7.1.3 Analysis and Results of MLR, RFR, GBR

To deepen our understanding of how the parameters $T_{\max}$, $T_{\min}$, and the number of steps influence both the final energy and the time taken by the simulated annealer, we employed machine learning techniques to analyse the data. Specifically, we used Multiple Linear Regression, Random Forest Regressor, and Gradient Boosting Regressor models. These models were chosen for their ability to capture both linear and non-linear relationships between the features and the labels.

1. **Multiple Linear Regression**: This model was used to quantify the linear relationships between the parameters and both the final energy and time taken. The coefficients of the model indicate the change in these metrics for a one-unit change in each parameter, holding all other parameters constant.

2. **Random Forest Regressor**: This ensemble learning method was employed to capture any non-linear relationships between the parameters and both the final energy and time taken. The feature importances provided by the model indicate the relative importance of each parameter in predicting these metrics.

3. **Gradient Boosting Regressor**: Similar to the Random Forest, this model also captures non-linear relationships but does so by optimising a loss function. The feature importances from this model further validate or challenge the findings from the Random Forest model.

By employing these machine learning models, we can better understand the intricate relationships between the simulated annealing parameters and the performance metrics. This analysis is invaluable for fine-tuning the algorithm for specific scenarios.

The results from the machine learning models provide valuable insights into the relative importance of the parameters $T_{\max}$, $T_{\min}$, and the number of steps in determining both the final energy and the time taken by the simulated annealer.

Table 1: Machine Learning Analysis Results for Simulated Annealer Parameters $(\lambda 0, \lambda 1, \lambda 2)$

Metric	**Linear Regression Coefficients**	**Random Forest Feature Importance**	**Gradient Boosting Feature Importance**
Final Energy	$[1.96 \times 10^{-3}, -26.79, -1.88 \times 10^{-2}]$	$[0.148, 0.139, 0.713]$	$[0.077, 0.070, 0.853]$
Time Taken	$[8.45 \times 10^{-4}, 1.37, 2.29 \times 10^{-3}]$	$[0.188, 0.214, 0.598]$	$[0.097, 0.167, 0.736]$

Analysis for Final Energy 1. **Linear Regression Coefficients**: The coefficients for $T_{\max}$, $T_{\min}$, and steps are approximately 1.96×10^{-3}, -26.79, and -1.88×10^{-2}, respectively. This suggests that $T_{\min}$ has a strong negative linear relationship with the final energy, while $T_{\max}$

and steps have a weaker positive and negative relationship, respectively.

2. **Random Forest Feature Importance**: The importances for $T_{\max}$, $T_{\min}$, and steps are 0.148, 0.139, and 0.713, respectively. This indicates that the number of steps is the most crucial parameter for determining the final energy, followed by $T_{\max}$ and $T_{\min}$.

3. **Gradient Boosting Feature Importance**: Similar to the Random Forest model, the importances are 0.077, 0.070, and 0.853 for $T_{\max}$, $T_{\min}$, and steps, respectively. This further confirms that the number of steps is the most significant factor.

Analysis for Time Taken 1. **Linear Regression Coefficients**: The coefficients for $T_{\max}$, $T_{\min}$, and steps are approximately 8.45×10^{-4}, 1.37, and 2.29×10^{-3}, respectively. All coefficients are positive, indicating a direct relationship between these parameters and the time taken.

2. **Random Forest Feature Importance**: The importances for $T_{\max}$, $T_{\min}$, and steps are 0.188, 0.214, and 0.598, respectively. Unlike the final energy, the time taken is more evenly influenced by all three parameters, with steps still being the most important but not overwhelmingly so.

3. **Gradient Boosting Feature Importance**: The importances are 0.097, 0.167, and 0.736 for $T_{\max}$, $T_{\min}$, and steps, respectively. This aligns with the Random Forest results, confirming that while steps are important, $T_{\max}$ and $T_{\min}$ also play a significant role.

In summary, the number of steps appears to be the most critical parameter for both the final energy and the time taken, but its influence is more pronounced for the final energy. The $T_{\min}$ parameter also shows a strong, albeit negative, linear relationship with the final energy. These insights can guide the fine-tuning of the simulated annealing algorithm for specific applications.

7.1.4 Correlation Statistical Analysis and Results

To further deepen our understanding of the model's behavior, we conducted advanced statistical analysis using Pearson correlation coefficients. This method allows us to quantify the linear relationship between two variables, providing a value between -1 and 1.

Time Taken Analysis Initially, we focused on understanding how the time taken for the algorithm to run is influenced by the parameters $T_{\max}$, $T_{\min}$, and *steps*. We filtered the data for unique combinations of $T_{\min}$ and *steps*, and calculated the Pearson correlation with $T_{\max}$. We repeated this process for the other parameters as well. The average correlations were then computed to provide a summary statistic.

Final Energy and Convergence Time Analysis We extended this analysis to two other key metrics: the final energy (or cost) and the convergence time of the algorithm. Similar to the time taken, we calculated the Pearson correlation coefficients for these metrics against each of the parameters $T_{\max}$, $T_{\min}$, and *steps*.

What This Adds to the Data Analysis This advanced statistical analysis provides several layers of insight:

1. **Parameter Importance**: By looking at the strength and direction of the correlation, we can identify which parameters are most influential in determining the time taken, final energy, and convergence time.

2. **Optimisation Guidance**: Understanding these correlations can guide us in fine-tuning the parameters for specific outcomes, such as minimizing time or energy.
3. **Model Robustness**: The analysis also helps in understanding how sensitive the model is to changes in each parameter, which is crucial for assessing the model's robustness and reliability.

By combining machine learning models with statistical correlation analysis, we obtain a comprehensive view of how our simulated annealing model behaves, which parameters are most critical, and where we might focus for further optimisation.

Table 2: Average Correlations for Different Metrics

Metric	$T_{\max}$	$T_{\min}$	Steps
Time Taken	-0.053	0.006	0.669
Final Energy	0.035	-0.067	-0.653
Convergence Time	-0.814	nan	nan

Time Taken Analysis

- **Tmax:** The average correlation between time taken and $T_{\max}$ is approximately -0.053. This is a very weak negative correlation, suggesting that increasing $T_{\max}$ has a negligible effect on reducing the time taken by the algorithm.
- **Tmin:** The average correlation between time taken and $T_{\min}$ is approximately 0.006, which is close to zero. This indicates that $T_{\min}$ has virtually no effect on the time taken by the algorithm.
- **Steps:** The average correlation between time taken and steps is approximately 0.669. This is a moderate positive correlation, indicating that as the number of steps increases, the time taken by the algorithm also tends to increase.

Final Energy Analysis

- **Tmax:** The average correlation between final energy and $T_{\max}$ is approximately 0.035, which is a very weak positive correlation. This suggests that $T_{\max}$ has a negligible effect on the final energy.
- **Tmin:** The average correlation between final energy and $T_{\min}$ is approximately -0.067, which is a very weak negative correlation. This suggests that increasing $T_{\min}$ has a negligible effect on reducing the final energy.
- **Steps:** The average correlation between final energy and steps is approximately -0.653. This is a moderate negative correlation, indicating that as the number of steps increases, the final energy tends to decrease.

Convergence Time Analysis

- **Tmax:** The average correlation between convergence time and $T_{\max}$ is approximately -0.814, which is a strong negative correlation. This suggests that increasing $T_{\max}$ significantly reduces the convergence time.
- **Tmin and Steps:** The correlation is *nan*, which means that the data is insufficient to establish a correlation. This could be due to a lack of variability in the data for these parameters

Effect of Zero Variance on Correlation Upon conducting a more detailed analysis, it was observed that several combinations of parameters exhibited zero variance for the metric of convergence time. Specifically, zero variance was detected for combinations involving $T_{\max}$ values of 50000, 25000, 10000, and 15000 with various $T_{\min}$ and steps values.

The absence of variance in these specific scenarios implies that the convergence time remained constant across multiple runs of the simulated annealer. This lack of variability in the data led to an undefined correlation coefficient, represented as *nan* in our results.

In statistical terms, a zero variance indicates that the data points are identical, making it impossible to establish a linear relationship with other variables. Therefore, the correlation coefficient could not be computed for these combinations. This inturn affects the overall interpretability of the correlation analysis for convergence time with respect to $T_{\min}$ and steps.

Interpretation

- **Time Taken:** The number of steps (steps) is the most influential factor affecting the time taken by the algorithm.
- **Final Energy:** The number of steps (steps) is also the most influential factor affecting the final energy, but in the opposite direction: more steps lead to lower energy.
- **Convergence Time:** The maximum temperature ($T_{\max}$) is the most influential factor affecting the convergence time, with higher values leading to faster convergence.

Does it Make Sense?

- The strong influence of steps on both time taken and final energy makes sense, as more steps would naturally require more computational time but also allow for a more refined solution, leading to lower energy.
- The strong negative correlation between $T_{\max}$ and convergence time also makes sense. A higher $T_{\max}$ would mean that the algorithm is more willing to explore the solution space quickly, leading to faster convergence.

Conclusion This detailed analysis provides valuable insights into how each parameter influences the performance metrics, aiding in the fine-tuning of the algorithm for different scenarios.

7.1.5 Visualisation of results

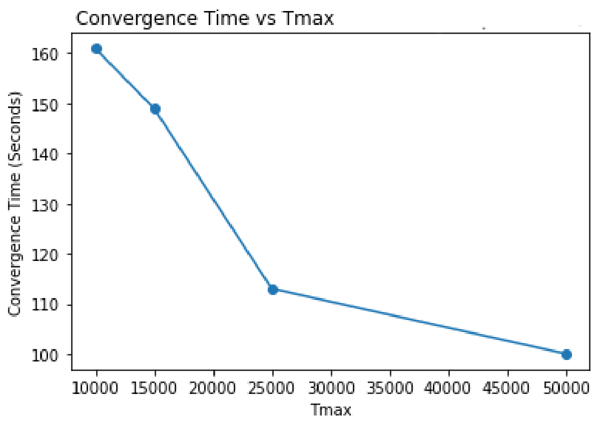

Figure 12: Convergence Time Vs Tmax Average

The graph above depicts the average trend across all Convergence Time vs Tmax combinations of Tmin and Steps. This clearly gives a good visual representation of the results found earlier. There is a strong correlation between convergence time decreasing and Tmax temperature.

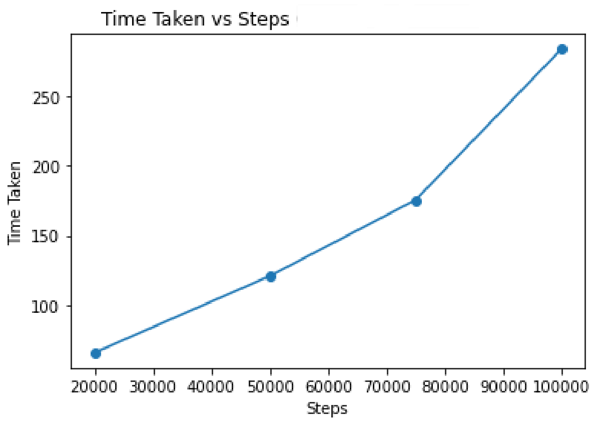

Figure 13: Time Taken Vs Steps Average

The graph above depicts the average trend across all Time Taken Vs Steps across all combinations of Tmin and Tmax. A strong positive correlation can be seen between number of steps and the time taken for the algorithm to complete it's search. Validating the correlation we saw earlier.

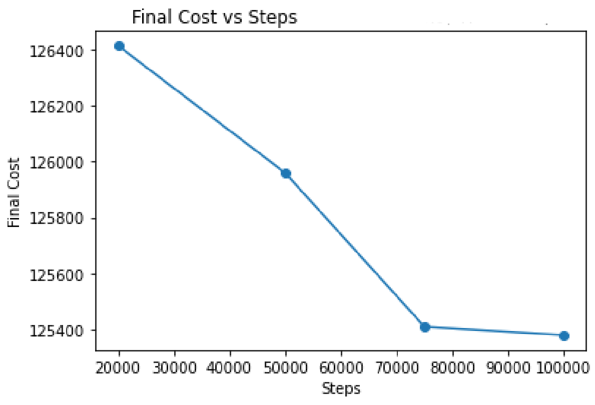

Figure 14: Final Cost Vs Steps Average

The graph above depicts the average trend across all Final Cost Vs Steps across all combinations of Tmin and Tmax. As witnessed in the data earlier, the correlation was negative as can be seen here. This gives us some visual affirmation of our results.

7.1.6 Overview of Results

The comprehensive analysis conducted on the final model revealed key insights into the performance and efficiency of the simulated annealing algorithm. Various metrics such as final energy, time taken, and convergence time were scrutinised under different configurations of T_{max}, T_{min}, and steps.

Optimal Values Our analysis suggests that the parameter 'steps' holds the most influence on both the time taken and the final energy, with high positive and negative correlations respectively. T_{max} and T_{min} showed less impact, indicating that a moderate range of these temperatures could be sufficient for effective optimisation. Specifically, larger 'steps' values are recommended for quicker convergence and lower final energy, while the choice of $T_{\max}$ and $T_{\min}$ can be more flexible. However, $T_{\max}$ did see a strong correlation with convergence time, indicating that a broader sweep of the solution space could lead to faster convergence.

Zero Variance Impact It's important to note that the correlation analysis for convergence time was affected by zero variance in several parameter combinations. This suggests that for those specific configurations, the algorithm's performance was invariant, making it challenging to establish any meaningful relationships.

Future Improvements While the analysis provides a robust starting point for parameter selection, it is not without limitations. The issue of zero variance, for instance, could be mitigated by introducing more variability in the initial conditions or by employing a more complex cost function. Additionally, further studies could explore the impact of other hyperparameters or employ machine learning techniques to dynamically adjust parameters during the optimization process.

Final Remarks In summary, this analysis serves as a foundational guide for selecting appropriate parameters for the simulated annealing algorithm. It highlights the importance of the 'steps' parameter and provides a nuanced understanding of how $T_{\max}$ and $T_{\min}$ can be configured for different optimisation scenarios.

7.2 Cost Function λ analysis

7.2.1 Framework

The terrain chosen is not excessively important in this analysis. Albeit we want a landscape which can clearly showcase the variability of paths when biased to different constraints. The 'S' shaped valley is chosen as it offers an interesting culmination on elevation and gradients.

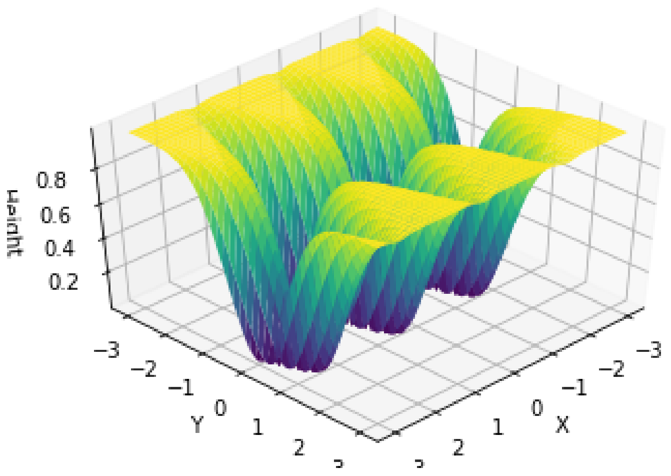

Figure 15: λ analysis terrain

7.2.2 Sensitive Sensor focused Model

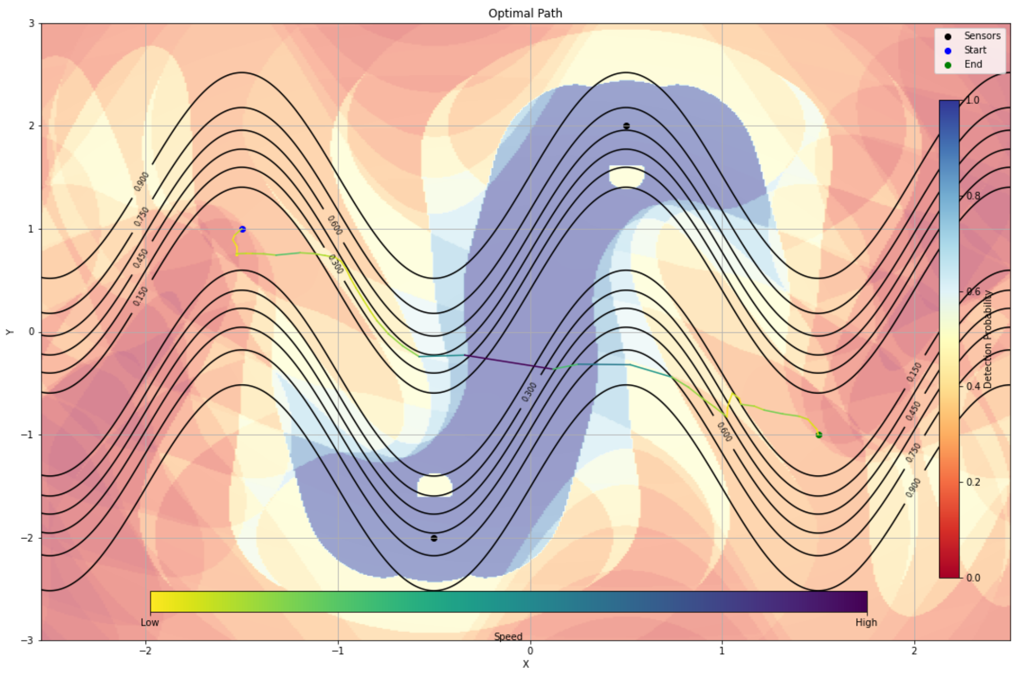

Figure 16: $\lambda_0 = 10000, \lambda_1 = 10, \lambda_2 = 10$

The path shows little regard for gradient as it traverses across high and low elevation freely. The focus, for the model, lies on crossing the high detection zone. This is where the velocity increases and the path is quite perpendicular to the detection zone.

7.2.3 Sensitive Velocity focused Model

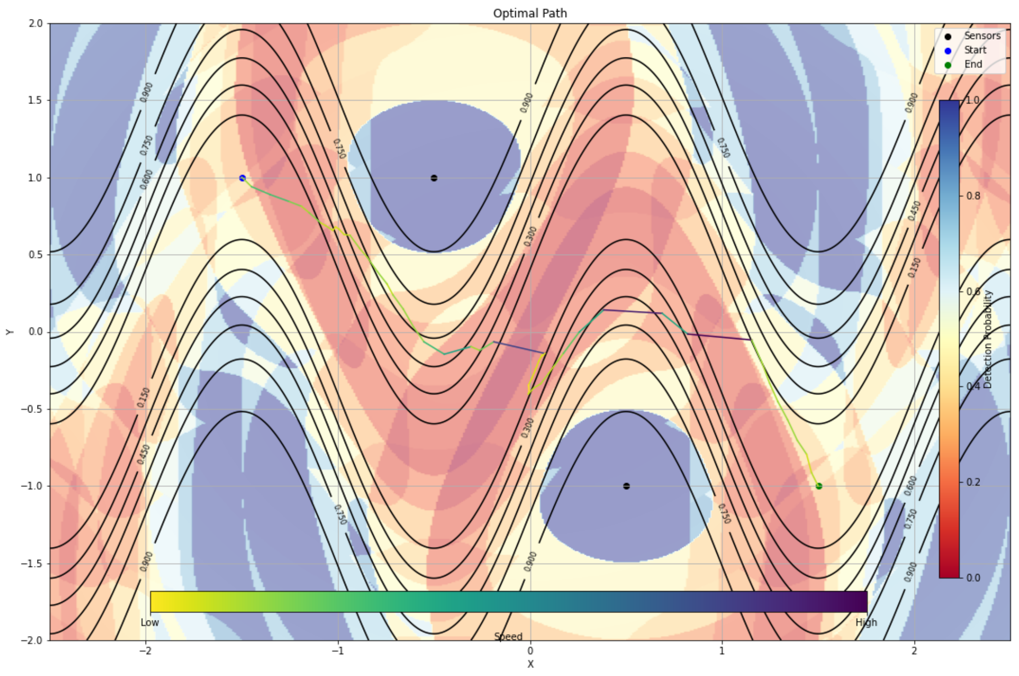

Figure 17: $\lambda_0 = 10, \lambda_1 = 10000, \lambda_2 = 10$

A shortcoming of this particular model is the chaotic nature of the path. In the future adjusting the step values to a larger number or a lower Tmin may have helped the cause. We can still see the path avoiding the elevation of the terrain to a large extent. The path mainly tries to stick to the lowest elevation areas of the map to avoid increase in gradient.

7.2.4 Sensitive Time focused Model

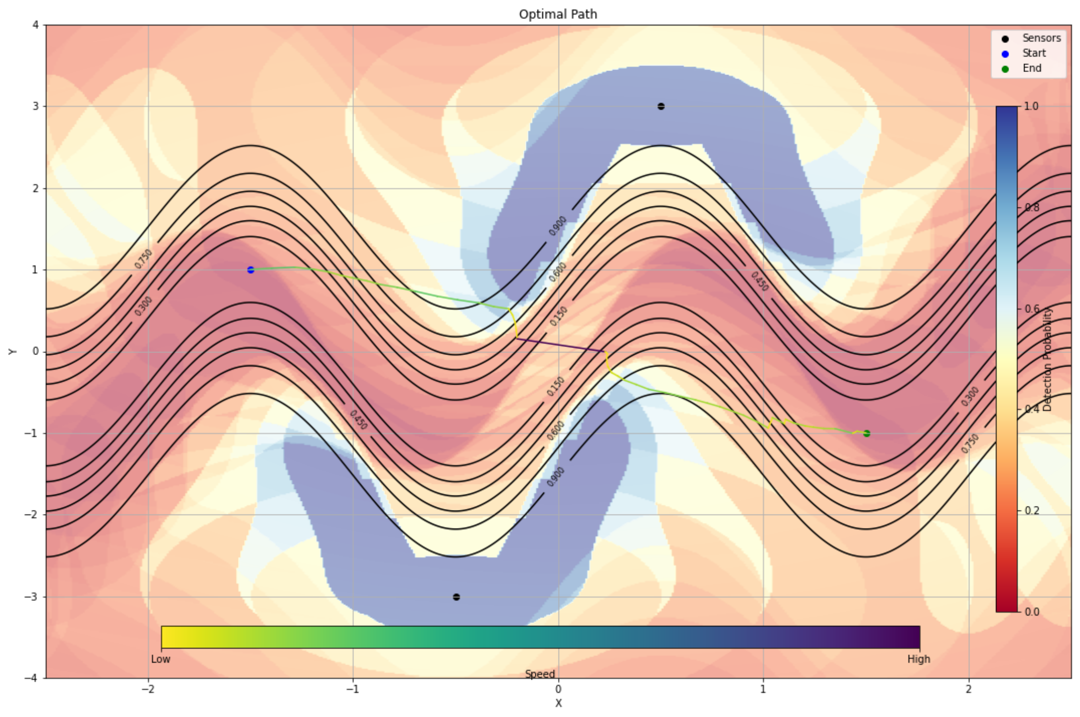

Figure 18: $\lambda_0 = 10, \lambda_1 = 10, \lambda_2 = 10000$

This graph demonstrates perfectly a sensitive time focused model. The path that can be seen demonstrates little care for gradients or sensors. The main focus is making the path as time efficient as possible, hence the straight-line nature of the path.

8 Real-world Application: Mountain Rescue Scenario

In this section, we present a real-world application of our algorithm. The focus is to demonstrate its adaptability by adjusting parameters and constraints for a specific scenario. The careful consideration of these adjustments is of paramount importance to address the unique challenges posed by the scenario at hand.

8.1 Description of the Scenario

Imagine a harrowing situation where an explorer finds themselves stranded in a treacherous, snowy mountain environment during an expedition. Recognising the urgency of the situation, the explorer contacts the authorities. A mountain rescue team is deployed to save the stranded individual.

The weather in the region is extremely dangerous, characterised by fierce storms. This renders helicopter operations infeasible near the individual's location. Compounding the challenge, a towering mountain peak obstructs the direct path of the rescuers.

Perhaps the most critical consideration is the presence of unstable pockets of snow scattered throughout the terrain. Triggering an avalanche in these areas could jeopardise the safety of the rescuers, the stranded individual and other mountaineers in the area. Conversley, while care must be taken when traversing the path, time is off the escence. It is imperative to reach the explorer with haste as Navigating these treacherous regions is imperative to ensure a successful rescue operation while minimising the risks involved.

8.2 Creating the Scenario-Specific Model

In this subsection, we describe the process of creating a scenario-specific model for the mountain rescue scenario.

8.2.1 Mapping the terrain

The initial step involves mapping the landscape, which entails acquiring a high-resolution height map of the area. This height map is typically obtained from real-world data sources and represents the elevation variations across the terrain. The goal is to transform this height map into a format that can be seamlessly integrated into the model.

The following code snippet illustrates this process:

```
# Load the image and convert to grayscale
img = Image.open('HeightMapPNG.png').convert('L')

# Convert the image to a numpy array and normalise
data = np.array(img)
data = data / np.max(data)

# Create an x, y coordinate grid for the original image
x = np.linspace(0, 1, data.shape[1])
y = np.linspace(0, 1, data.shape[0])

# Create an interpolating function
f = interp2d(x, y, data, kind='cubic')
```

In this example, a height map of a real mountainous area has been obtained [8], with the elevation data stored in a PNG file. We load this image and convert it to grayscale before putting it into a numpy array format for further processing. Normalisation is applied to ensure that the elevation values are within a suitable range for modeling.

To create a continuous representation of the terrain, we create an x, y coordinate grid that corresponds to the original image. The heart of this process lies in the interpolation step. We use the 'interp2d' function with a cubic interpolation method. This generates a continuous function that provides elevation values for any given coordinate within the original image.

```
def Height(Position):
    x, y = Position[0], Position[1]
    h = f(x, y)[0]  # f returns a 1x1 array, so we extract the single value
    return [x, y, h]
```

Finally, we recall and adjust the 'Height' function we developed earlier. We replace the previous function with our new real-terrain function.

8.2.2 Creating the path

In this section, we discuss the adjustments made to the parameters and constraints in order to accurately represent the environment and scenario for the mountain rescue path. The goal is to ensure that the simulated path aligns with the specific challenges and priorities of the mountain rescue scenario.

First, in the `PropSens` function, the `sig` parameter is increased from `sig = 4` to `sig = 6`. This adjustment increases the sensitivity of the sensors to proximity. Given that one of the main priorities is to avoid triggering an avalanche, increasing the `sig` value emulates this sensitivity and ensures that the path remains clear of the trigger locations.

Next, in the `ProbDetect` function, the detection method based on Line of Sight (LoS) is removed. The trigger is now solely modeled based on proximity, and LoS no longer affects it. The line of code `probs.append(PropSens(dist)*LineOfSightResistance(Point, Sensor))` is modified to `probs.append(PropSens(dist))`.

To define the time constraint on the path, the `TotalT_Thresh` parameter is adjusted. The `Min_Speed` is increased to 3. This modification reflects the urgency of the scenario, requiring the mountaineers to move at a faster pace.

The `RunAnnealer` function is also adjusted with the earlier analysis in mind. A high `initial_temp` value of 50000 is set to explore the solution space effectively and decrease convergence time. Additionally, the `total_steps` value is chosen to produce a low final cost while ensuring computational efficiency.

Regarding the specific values used for ‘start‘ and ‘end‘ locations:

```
# StartEnd
start = [-1.5, 1]  % Helicopter drop-off point
end = [1.5, -1]  % Explorer’s location
sensors = [[0.5, 2], [-0.5, -2]]
lambda0 = 10000
lambda1 = 1000
lambda2 = 10000
lambda3 = 2
```

The sensor locations are strategically placed at the avalanche trigger locations to simulate the critical areas. The λ values are skewed to prioritise the avoidance of the sensors and the time efficiency of the route, aligning with the goals and constraints of the mountain rescue scenario.

8.2.3 Running the Model

The model took a total of 12 minutes and 13 seconds to run. The generated path, as shown in Figure 20, exhibits several noteworthy characteristics.

The plot is very detailed, accurately depicting the high probability of detection areas. The graph also displays the elevation well with contour lines showing the high elevation areas.

Firstly, the path effectively steers clear of the avalanche trigger locations while maintaining a direct route. This is crucial for ensuring the safety of the rescuers and the individual in need. It adapts its speed strategy, moving swiftly through the relatively safe initial section of the path. The agents then slowing down near the end as it approaches the second trigger location,

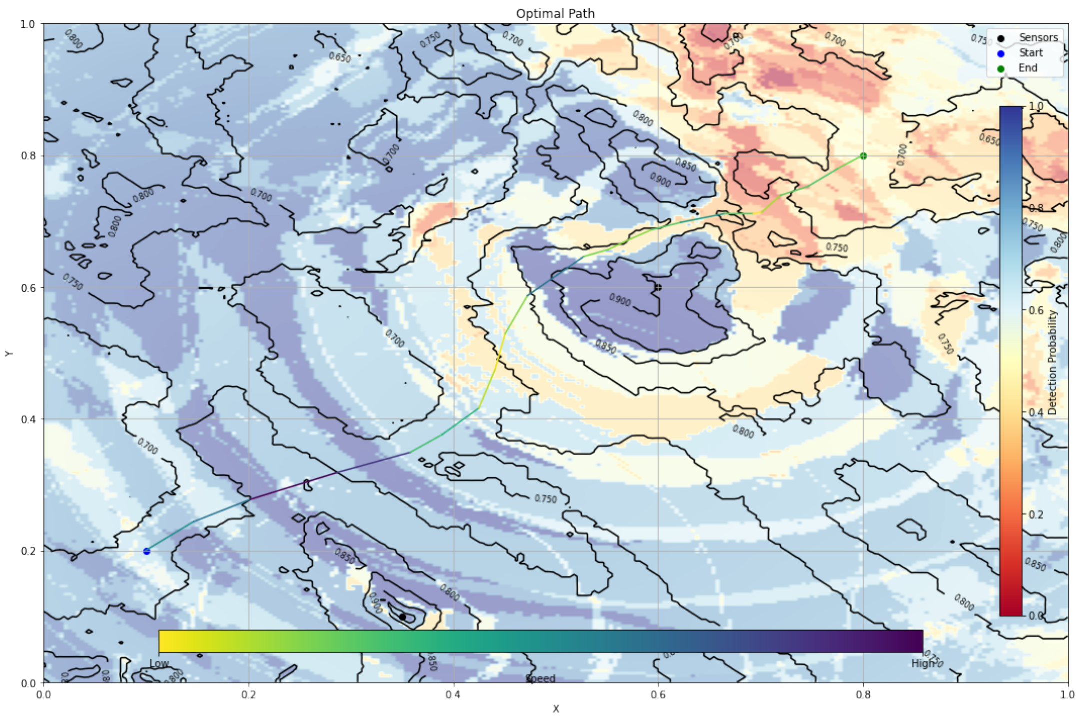

Figure 19: Path Generated for Mountain Rescue Scenario

ensuring careful traversal.

Additionally, the path avoids the peak of the mountain, further enhancing safety and efficiency. This avoidance of obstacles, both natural and scenario-specific, showcases the adaptability and intelligence of the generated path.

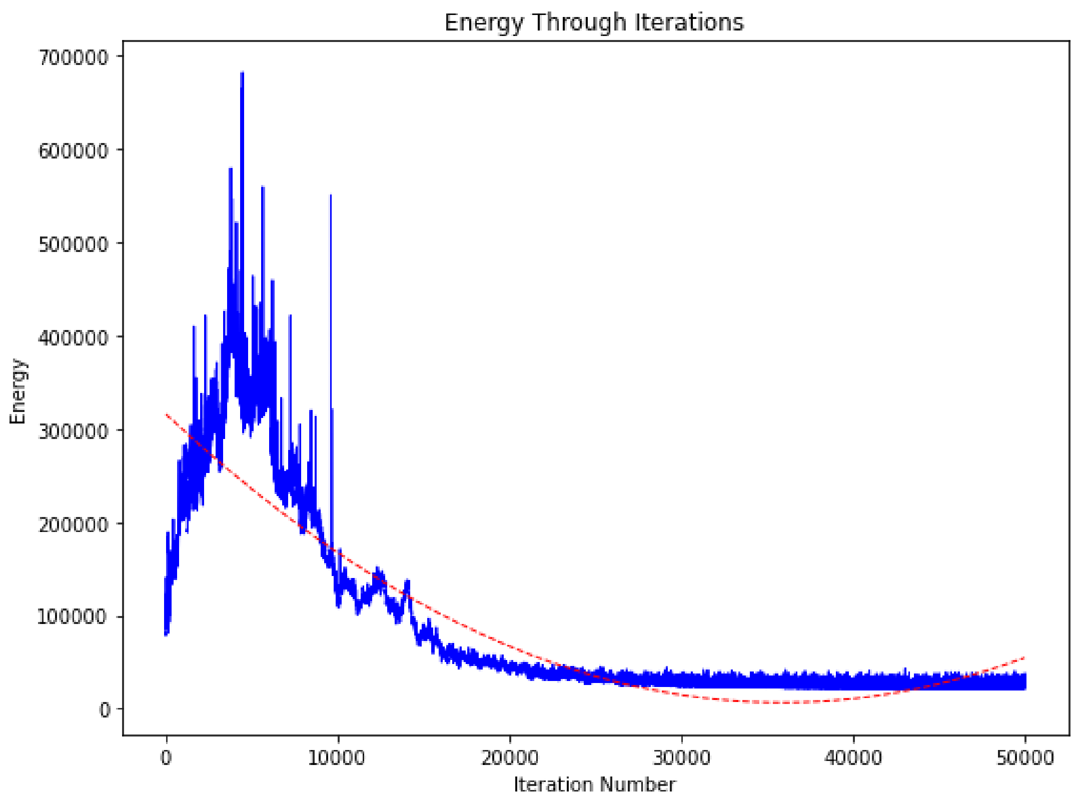

Figure 20: Energy plot for Path Generated for Mountain Rescue Scenario

The energy plot of the optimisation process demonstrates that the algorithm explored the solution space effectively and converged rapidly to a low final cost. This outcome signifies a successful application of the algorithm, avoiding local minima and efficiently finding an optimal path.

In summary, the path generated by the model in this mountain rescue scenario serves as a perfect demonstration of the algorithm's capabilities. It successfully addresses the specific constraints and priorities of the scenario, setting the stage for its future use in real-world applications.

9 Other Potential Applications

9.1 Supply Chain

Given the algorithm's versatility it holds significant promise in supply chain management. It could be used to optimise routes for goods transportation, inventory management, and even in supplier selection processes. The algorithm's computational efficiency makes it a viable option for real-time supply chain optimisations.

9.2 Security Systems

Given that this report is titled "Escape the Sensors," the algorithm's applicability in optimising sensor networks for security is evident. It could be used to optimise the placement and sensitivity settings of sensors in a security network. This would be particularly useful in scenarios that require bypassing sensors without triggering alarms.

10 Conclusion

The dissertation, titled "Escape the Sensors," initiated with the objective of addressing a complex problem in pathfinding. The primary goal was to develop an algorithmic solution that is not only effective but also realistic in its application.

10.1 Literature Review and Research Direction

An exhaustive literature review was conducted to understand the current state of research in the field. This exercise ensured that the study was novel, avoiding repetition and instead expanding upon existing research. The review influenced the algorithmic choices made, offering inspiration and techniques that guided the research direction.

10.2 Problem Decomposition

Once the research direction was secure, the problem was meticulously decomposed, laying out the challenges and complexities involved. The mathematical aspects were tackled first, providing a solid foundation for the algorithmic development.

The simulated annealing algorithm is juxtaposed with other algorithms like SGD and Genetic algorithms for comparative evaluation. While Genetic algorithms necessitate large population sizes and SDG has shortcomings in escaping local optima, simulated annealing presents a more computationally efficient alternative.

10.3 Choice of Method and Mathematical Fundamentals

Simulated annealing was chosen as the most suitable method for solving the problem at hand. The algorithm was broken down to its mathematical fundamentals to ensure a comprehensive understanding before its implementation.

10.4 Model Development

The research culminated in the development of a 2D model, where the mathematical foundations were translated into code. The 2D model served as the foundation for the algorithm and was followed by a more intricate 3D model.

10.5 Model Analysis

An in-depth analysis of the final model was conducted, providing both numerical and qualitative measures of the algorithm's performance.

The quantitative analysis too place in the parameter tuning section. Here, the pivotal role played by the parameters Tmax, Tmin, and steps in the simulated annealing algorithm were assessed. The number of steps emerged as the most critical parameter, influencing both the final energy state and the computational time required by the algorithm. Additionally, Tmax and Tmin also contribute significantly, with Tmin exhibiting a strong, albeit negative, linear relationship with the final energy. The convergence time however, was highly correlated to the Tmax value. Despite some shortcomings like zero variance affecting the analysis, the evaluation was comprehensive and these parameters serve as key levers in fine-tuning the algorithm for specialised applications.

The lambda analysis section focused on the qualitative aspects, using different lambda values to showcase the algorithm's adaptability to the changing cost function. The difference was evident, with the lambda values having a real effect on the outcome of the paths. With each value biasing the path to sensitivity to different constraints.

10.6 Real-world Demonstration

The algorithm was then applied to a real-world scenario. The demonstration was successful, indicating that the algorithm would perform effectively in a real-life setting.

10.7 Realism and Final Remarks

Given that the initial problem was to make the solution as realistic as possible, the dissertation successfully achieves this objective. The algorithm not only models real-life scenarios accurately but has also been rigorously tested to prove its efficacy. The research serves as a seminal work in the field, offering valuable insights and setting a new standard for realism in algorithmic solutions.

11 Recommendations for Future Work

11.1 Parameter Tuning

Future studies should focus on fine-tuning the parameters Tmax, Tmin, and steps to optimise the algorithm for specialised applications. This is crucial for enhancing its versatility and applicability.

11.2 Algorithmic Comparisons

Comparative studies with other optimisation algorithms, such as Genetic Algorithms. This could provide a more comprehensive understanding of the algorithm's strengths and weaknesses.

11.3 Real-world Testing

Pilot studies in real-world scenarios are recommended to validate the algorithm's efficacy. This would provide empirical evidence to support its theoretical capabilities.

11.4 Statistical Methods

Other statistical methods could be employed to validate the algorithm's performance further. This would add another layer of credibility to the research findings.

Acknowledgments

References

[1] Zeyad Abd Algfoor, Mohd Shahrizal Sunar, and Hoshang Kolivand. A comprehensive study on pathfinding techniques for robotics and video games. *International Journal of Computer Games Technology*, 2015:11, 2015.

[2] Anton Andreychuk, Konstantin Yakovlev, Pavel Surynek, Dor Atzmon, and Roni Stern. Multi-agent pathfinding with continuous time. *Artificial Intelligence*, 305:103662, 2022.

[3] Mark Fleischer. Simulated annealing: Past, present, and future. In *Proceedings of the 27th Conference on Winter Simulation*, WSC '95, page 155–161, USA, 1995. IEEE Computer Society.

[4] Kemal Ihsan Kilic and Leonardo Mostarda. Optimum path finding framework for drone assisted boat rescue missions. In Leonard Barolli, Isaac Woungang, and Tomoya Enokido, editors, *Advanced Information Networking and Applications*, pages 219–231, Cham, 2021. Springer International Publishing.

[5] Daniel R. Lanning, Gregory K. Harrell, and Jin Wang. Dijkstra's algorithm and google maps. In *ACM SE '14*, Kennesaw, GA, USA, 2014.

[6] Masato Noto and Hiroaki Sato. A method for the shortest path search by extended dijkstra algorithm. In *Proceedings of the IEEE Conference*, pages 2316–2320, 2000.

[7] PerryGeo. Python module for simulated annealing, 2023. Accessed: 2023-09-11.

[8] Motion Forge Pictures. Landscape height maps, 2023. Accessed: 2023-09-11.

[9] Xing Wu Jun Qi Qidong Han, Shuo Feng and Shaowei Yu. Retrospective-based deep q-learning method for autonomous pathfinding in three-dimensional curved surface terrain. *Appl. Sci.*, 13:6030, 2023.

[10] Alexey Skrynnik, Alexandra Yakovleva, Vasilii Davydov, Konstantin Yakovlev, and Aleksandr I. Panov. Hybrid policy learning for multi-agent pathfinding. *IEEE Access*, 9:126034–126047, 2021.

[11] R. Somma, S. Boixo, and H. Barnum. Quantum simulated annealing, 2007.

[12] Miao Wang and Hanyu Lu. Research on algorithm of intelligent 3d path finding in game development. In *2012 International Conference on Industrial Control and Electronics Engineering*, pages 1738–1742, 2012.

[13] Sichen Xiao, Xiaojun Tan, and Jinping Wang. A simulated annealing algorithm and grid map-based uav coverage path planning method for 3d reconstruction. *Electronics*, 10:853, 2021.

[14] Imants Zarembo and Sergejs Kodors. Pathfinding algorithm efficiency analysis in 2d grid. *ENVIRONMENT. TECHNOLOGIES. RESOURCES. Proceedings of the International Scientific and Practical Conference*, 2(0):46–50, 2015.

[15] A Žilinskas. A review of statistical models for global optimization. *J Glob Optim*, 2:145–153, 1992.

A Appendix A: Supplementary Material

The Final Model code is as follows:

```
import os
import numpy as np
from simanneal import Annealer
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.colors import LinearSegmentedColormap, Normalize
import random

def PropSens(dist, sig = 15):
    p = 1/(1+((dist/sig)**2))
    return p
```

```
def ProbDetect(Sensors, Point):
    Point = Height(Point)
    probs = []
    for Sensor in Sensors:
        Sensor = Height(Sensor)
        dist = np.linalg.norm(np.array(Sensor) - np.array(Point))
        probs.append(PropSens(dist)*LineOfSightResistance(Point, Sensor))
    probs_sorted = sorted(probs, reverse=True)
    p1 = probs_sorted[0]
    p2 = probs_sorted[1]
    P = p1*p2
    return P

def LineOfSightResistance(Position, Sensor, interval = 0.1, strength = 0.8): #0<=strength<=
    Position = Height(Position)
    Sensor = Height(Sensor)
    dist = np.linalg.norm(np.array(Sensor) - np.array(Position))
    if dist == 0:
        Resistance = 0
        return Resistance
    Direc = (np.array(Sensor) - np.array(Position))/dist
    Lambda = 0
    CheckP = 0
    NLoSCheckP = 0
    while Lambda <= dist:
        CheckP += 1
        L = Position + Lambda*Direc
        HeightL = L[2]
        HeightS = Height([L[0],L[1]])[2]
        if HeightS > HeightL:
            NLoSCheckP += 1
        Lambda += interval
    NLoSPercentage = NLoSCheckP/CheckP
    Resistance = (-strength*NLoSPercentage)+1
    return Resistance**2

def StartSolPoints(Start, End):
    points = []
    direc = np.array(End)-np.array(Start)
    dist = np.linalg.norm(direc)
    numPoints = (20)+1
    round(dist)
    stepSize = dist/numPoints
    for Lambda in range(numPoints+1):
        point = Start + stepSize*Lambda*(1/dist)*direc
        points.append(point)
    return points

def Points2VelocityNGrad(Points):
    V = []
    Grads = []
    numVelocity = len(Points)-1
    for i in range(numVelocity):
        Start = np.array(Height(Points[i]))
        End = np.array(Height(Points[i+1]))
        Vector = End-Start
        L = np.sqrt(Vector[0]**2 + Vector[1]**2)
        Z = Vector[2]
        Grad = Z/L
        V.append(Vector)
        Grads.append(Grad)
    return V, Grads

def Plot(Start, End, Points, Sensors, Speeds):
    Start = np.array(Start)
    End = np.array(End)
    Sensors = np.array(Sensors)
    # Create a figure and axes with larger size
    fig, ax = plt.subplots(figsize=(15, 10))

    # Define the optimized path as a list of points
    optimized_path = Points

    # Extract x and y coordinates from the optimized path
    x_coords = [point[0] for point in optimized_path]
    y_coords = [point[1] for point in optimized_path]

    # Define the colors for the path based on speeds
    speed_values = np.array(Speeds)
    normalized_speeds = (speed_values - np.min(speed_values)) / (np.max(speed_values) - np.r
    cmap_speed = plt.cm.get_cmap('viridis_r')
    colors_speed = cmap_speed(normalized_speeds)

    # Compute the range of x and y coordinates for the entire graph
    x_min = min(np.min(x_coords), np.min(Sensors[:, 0]), Start[0], End[0])
    x_max = max(np.max(x_coords), np.max(Sensors[:, 0]), Start[0], End[0])
    y_min = min(np.min(y_coords), np.min(Sensors[:, 1]), Start[1], End[1])
    y_max = max(np.max(y_coords), np.max(Sensors[:, 1]), Start[1], End[1])

    # Add space to the x and y coordinate ranges
    x_range = np.linspace(x_min - 1, x_max + 1, 100)
    y_range = np.linspace(y_min - 1, y_max + 1, 100)
     # Plot the sensors
    ax.scatter(Sensors[:, 0], Sensors[:, 1], color='black', label='Sensors')

    ax.scatter(Start[0], Start[1], color='blue', label='Start')
    ax.scatter(End[0], End[1], color='green', label='End')

    # Plot the path with different colors based on speeds
    for i in range(len(x_coords) - 1):
        ax.plot([x_coords[i], x_coords[i + 1]], [y_coords[i], y_coords[i + 1]], color=colors
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_title('Optimal Path')
    ax.grid(True)

    # Add legends for sensors, start, and end
    ax.legend(loc='upper right', bbox_to_anchor=(1, 1))

    # Add this right after defining the x_range and y_range in the Plot function:
    X, Y = np.meshgrid(x_range, y_range)

    H = np.zeros(X.shape)

    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            H[i, j] = Height((X[i, j], Y[i, j]))[2]

    Z = np.zeros(X.shape)

    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            Z[i, j] = ProbDetect(Sensors, (X[i, j], Y[i, j]))

    # Create a heatmap using the detection probabilities
    cmap_heatmap = plt.cm.RdYlBu
    ax.imshow(Z, extent=[x_min-1, x_max+1, y_min-1, y_max+1], origin='lower', aspect='auto',

    contours = ax.contour(X, Y, H, colors='black', linestyles='solid')
    ax.clabel(contours, inline=True, fontsize=8)


    # Add the heatmap colorbar
    cax_heatmap = fig.add_axes([0.92, 0.15, 0.02, 0.7])
    cbar_heatmap = plt.colorbar(plt.cm.ScalarMappable(cmap=cmap_heatmap), cax=cax_heatmap)
    cbar_heatmap.ax.set_ylabel('Detection Probability')


    # Create a custom colorbar for the speed scale
    cax_speed = fig.add_axes([0.15, 0.1, 0.7, 0.03])  # Adjust the position and size of the
    cbar_speed = plt.colorbar(plt.cm.ScalarMappable(cmap=cmap_speed), cax=cax_speed, orienta
    cbar_speed.set_ticks([0, 1])  # Set the colorbar ticks
    cbar_speed.set_ticklabels(['Low', 'High'])  # Set the tick labels
    cbar_speed.ax.xaxis.set_label_coords(0.5, -1)  # Adjust the position of the colorbar lal
    cbar_speed.ax.set_xlabel('Speed')  # Set the colorbar label
    plt.tight_layout()

    return fig

def PlotE(Iter_E):
    # Sample energy values for each iteration (replace with your own data)
    energy_values = Iter_E

    # Generate x-axis values for iterations
    iterations = range(1, len(energy_values) + 1)

    # Adjusting figure size and creating the figure and axis objects
    fig, ax = plt.subplots(figsize=(8, 6))

    # Plotting the line graph with thin lines
    ax.plot(iterations, energy_values, linestyle='-', color='b', linewidth=1)

    # Adding labels and title
    ax.set_xlabel('Iteration Number')
    ax.set_ylabel('Energy')
    ax.set_title('Energy Through Iterations')

    # Adding a curve of best fit (quadratic curve)
    best_fit_curve = np.polyfit(iterations, energy_values, 2)  # Fit a 2nd-degree polynomia
    curve_fit_values = np.polyval(best_fit_curve, iterations)
    ax.plot(iterations, curve_fit_values, linestyle='--', color='r', linewidth=1)

    # Display the line graph
    plt.tight_layout()

    return fig

# Example usage:
# Assuming you have your data in the list `energy_values`, call the function like this:
# figure = PlotE(energy_values)
# plt.show()  # To display the figure if needed

def update(frame, figures):
    # Clear the current axes
    plt.clf()

    # Get the current figure
    fig = figures[frame]

    # Convert figure to image
    fig.canvas.draw()
    image = np.frombuffer(fig.canvas.tostring_rgb(), dtype='uint8')
    image = image.reshape(fig.canvas.get_width_height()[::-1] + (3,))

    # Display the image
    plt.imshow(image)
    plt.axis('off')

def iterations(list_points, list_speeds, slicing=1000):
    figures = []
    for i in range(len(list_points[::slicing])):
        points = list_points[::slicing]
        speeds = list_speeds[::slicing]
        point = points[i]
        speed = speeds[i]
        figure = Plot(start, end, point, sensors, speed)
        figures.append(figure)
        plt.close()

    # Create a new figure
    fig, ax = plt.subplots()

    # Create the animation
    ani = animation.FuncAnimation(fig, update, frames=len(figures), fargs=(figures,), interv
    ani.save('animation.mp4', dpi=500, writer='ffmpeg')
    plt.close()

def TotalT_ThreshCalc(Start, End, Min_Speed = 1.5):
    Dist = np.linalg.norm(np.array(End) - np.array(Start))
    Time = Dist/Min_Speed
    return Time

iter_Vectors = []
iter_Points = []
iter_T = []
iter_E = []
iter_Grads = []

# Define the problem class
class SimulatedAnnealer(Annealer):
    def __init__(self, intial_temp):
        initial_state = self.random_state()  # Generate initial state
        super().__init__(initial_state)
        self.Tmax = initial_temp


    def move(self):
        # Iterate through each point in the state and modify its direction and magnitude
        Vectors = self.state[0]
        Points = self.state[1]
        T = self.state[2]

        i = random.randint(1, len(Points)-2)

        TotalT_Curr = TotalT_Thresh + 10

        while TotalT_Curr > TotalT_Thresh:

            # Randomly adjust the direction of the point by a small angle
            adjustTheta = 1
            angle = np.random.uniform(-np.pi*adjustTheta, np.pi*adjustTheta)  # Modify this
            # Randomly adjust the magnitude of the point by a small factor
            adjustR = 0.1
            distR = np.random.uniform(0, adjustR)  # Modify this range as needed


            # Update the point with the new polar coordinates
            Points[i] = Points[i] + np.array([distR*np.cos(angle),
                                              distR*np.sin(angle)])

            Vectors, Grads = Points2VelocityNGrad(Points)


            adjustT = 0.1
            magnitude_factorT = np.random.uniform(1-adjustT, 1+adjustT)
            T = T*magnitude_factorT

            TotalT_Curr = T*len(Vectors)

        iter_Vectors.append(Vectors)
        iter_Points.append(Points)
        iter_T.append(T)
        iter_Grads.append(Grads)

        self.state = Vectors, Points, T


    def energy(self):
        Vectors = self.state[0]
        Points = self.state[1]
        T = self.state[2]

        P = 0
        for Point in Points:
            P += ProbDetect(sensors, Point)

        S = 0
        for Vector in range(len(Vectors)):
            Grad = iter_Grads[-1][Vector]
            EXP = lambda3+(2/np.pi)*np.arctan(Grad)
            S += (np.linalg.norm(Vectors[Vector]))**EXP/T

        TotalT = T*len(Vectors)

        E = float(lambda0*P*T + lambda1*S + lambda2*TotalT)

        iter_E.append(E)

        return E


    def random_state(self):
        points = StartSolPoints(start, end)
        vectors, grads = Points2VelocityNGrad(points)
        iter_Grads.append(grads)
        T = TotalT_ThreshCalc(start, end, Min_Speed = 2.5)/len(vectors)
        return vectors, points, T

def RunAnnealer(Initial_temp = 25000):

    # Create an instance of SimulatedAnnealer
    annealer = SimulatedAnnealer(initial_temp)

    # Set initial state
    initial_state = annealer.random_state()

     # Run the annealing process
    best_state, best_energy = annealer.anneal()

    iter_Speeds = []
    for vectors in iter_Vectors:
        speeds = []
        for speed in vectors:
            speeds.append(np.abs(speed[0])/best_state[2])
        iter_Speeds.append(speeds)

    return best_state, best_energy, iter_Vectors, iter_Points, iter_T, iter_Speeds

def Visialise(Iter_Points, Iter_Speeds, Iter_E, slicing = 5000):
    length = len(iter_Points)
    points = iter_Points[length-1]
    speeds = iter_Speeds[length-1]
    iterations(iter_Points, iter_Speeds, slicing)
    a = Plot(start, end, points, sensors, speeds)
    b = PlotE(Iter_E)

def Height(Position):
    x = Position[0]
    y = Position[1]
    h = np.exp(-((x-2.5)**2+(y-2)**2)*2)*2+np.exp(-((x+1.5)**2+(y+1)**2)*2)*2
    return [x, y, h]

#StartEnd
start = [ -3, -1]
end = [ -0.2, 0.2]

#Sensors
sensors = [[-1.8, -0.8], [-1.5, -0.8]]
initial_temp = 25000
lambda0 = 10000
lambda1 = 100
lambda2 = 100
lambda3 = 2
```

```
TotalT_Thresh = TotalT_ThreshCalc(start, end, Min_Speed = 0.1)

best_state, best_energy, iter_Vectors, iter_Points, iter_T, iter_Speeds = RunAnnealer()

Visialise(iter_Points, iter_Speeds, iter_E, slicing = 5000)
```