

# Aerial Robotics Kharagpur Software Task Round 2021 Documentation

Ishan Manchanda

**Abstract**— This document details my solutions for the Task Round of the Aerial Robotics Kharagpur (ARK) Software Team selections 2021.

For Task 2.1 (Optimize Me), I leveraged parallelization, locality of reference, bottom-up dynamic programming and other such concepts to minimize the runtime of a short C++ program. These techniques are employed to differing extents in nearly all software to ensure that program runtimes don't grow arbitrarily and are within required bounds.

In Task 2.2 (Face Detection Game), I used OpenCV's Deep Neural Network (DNN) module along with a pre-trained Caffe model to detect and track the user's face, allowing them to play a simple ball bouncing game. Face detection is used in several fields such as photography for applications like auto-focus and has paved the way for the innumerable applications of face recognition.

In Task 3.1 (Path Planning and Computer Vision Puzzle), I used OpenCV to perform image processing tasks such as pattern matching and implemented path finding algorithms such as A\* and its specializations - Dijkstra's Algorithm and Best-First Search. I was able to successfully solve all levels of the puzzle and compare the performance of the various algorithms. Both computer vision and path planning are of immense importance to several fields, especially including autonomous robotics.

For Task 4.1 (Tracking Multiple Objects), I implemented and tuned a Kalman Filter that tracks the positions of six objects with respect to a known reference point. The tuning of the filter is flexible and generalizes across all three test cases with varying levels of noise and a final test case which includes erroneous values. The Kalman Filter and its variants (Extended, Unscented) are indispensable for many problems dealing with tracking and localization.

For Task 4.2 (3D Tic Tac Toe Agent), I implemented a minimax agent that optimally plays the standard, 2D version of Tic Tac Toe. Additionally, I partially implemented a Deep Q-Learning agent that plays the 3D extension of the game. Both the minimax algorithm and reinforcement learning algorithms like Deep Q-Learning are widely used in various games and simulations.

## INTRODUCTION

The problem statement comprises five tasks arranged in increasing order of complexity, with the first two being basic, the third intermediate, and the last two advanced. The tasks span a variety of fields and problems related to autonomous robotics, as mentioned above.

This document primarily contains discussions on the various approaches I used to solve the tasks, and compares them with alternatives wherever applicable. It also discusses implementation details where needed but doesn't cover the code written to solve the tasks. The code is separately available in git repositories and is well-commented. READMEs are included for each task for further clarifications about the project structure. The code should be fairly lucid for those

two reasons, and coupled with the discussions in this document it should provide a straightforward yet comprehensive understanding of my work.

My approaches for most of the tasks have been direct (mentioned in the problem statement/easily found while searching) but effective. Alternate approaches were considered, some flashier and some easier to implement, but the current approaches were adopted as a sweet-spot, taking into account considerations like computational requirement, time to learn and implement, accuracy or other performance metrics, etc.

Most of these approaches have been introduced in the abstract and will be discussed in detail under the relevant task.

## I. TASK 2.1: OPTIMIZE ME

### PROBLEM STATEMENT

The objective of this task is to minimize the runtime of a provided C++ program for increasingly large input sizes. A single parameter  $n$  determines this size and takes values as multiples of four between 4 and 1,000. The provided file has three distinct operations that can be optimized, along with the necessary setup and boilerplate.

The program starts by generating two  $n \times n$  matrices filled with random values, with the value being a 'cost' of that cell. The first operation it performs is computing the minimum (for the first matrix and maximum for the other) cost of reaching the bottom-right-most cell from each cell, with only right- and down-wards moves allowed. These values are computed using two recursive functions. The next step is computing the matrix product of the two cost matrices and storing that in a third matrix. Finally, a  $4 \times n$  filter is applied to the matrix which computes a dot product taking four rows at a time and generates a final  $\frac{n}{4} \times 1$  vector. The filter step is only partially implemented in the provided file and has to be completed.

### RELATED WORK

Minimum/maximum cost path problems are frequently encountered in software and there exist a myriad of algorithms that solve them. However, most of these algorithms like Floyd-Warshall (which computes all-pairs minimum-cost paths) and Bellman-Ford (which computes the least-cost paths from a single source to all other vertices) are extremely inefficient for grids. This is because grids are sparse graphs as the number of edges increases linearly with the number of vertices rather than quadratically.

In the special case of the problem presented here, where costs from all cells to a corner cell are to be computed and

the movement direction is restricted, Dynamic Programming provides the optimal solution.

Matrix multiplication and applying filters on matrices are widely used operations and thus have had a lot of research on optimization for matrices of large sizes [1], [2]. Furthermore, use of GPUs to speed up these operations has also been well-explored [3]. Some of these techniques have been employed in the final approach presented here and yet others have been implemented for comparison.

### INITIAL ATTEMPTS

With some background in competitive programming, the use of dynamic programming for the first part is obvious. The initial implementation used a top-down DP approach with a memo table for the computed costs.

For the second part, both the block matrix multiplication algorithm as well as Strassen's algorithm were explored as possible options, especially since  $n$  was given to be a multiple of four. Moreover, CPU-based parallelization was also used to speed up these algorithms.

Preliminary testing revealed that the third operation required relatively much lower computation time, and as such CPU-based parallelization seemed sufficient to bring the runtime to an acceptable level.

Due to prior experience, the initial approaches directly targeted optimizing for the largest input size of  $n = 1000$ .

### FINAL APPROACH

The initial attempt ended up being significantly modified after some testing. After receiving clarification that the program could be modified in any way (including the function signatures), the top-down memoization-based approach was replaced by a bottom-up tabulation-based one. This approach not only avoids the function call overhead associated with creating stack frames (this cost might be optimized away by modern compilers using function inlining), it also allows leveraging locality of reference to maximize the chance of cache hits and prefetching.

In the tabulation-based approach, the min and max costs are computed from right to left, and from bottom to top. In a competitive programming scenario, a common approach is to fill the last row and column at the beginning and then to iterate up from the second-last row. However, the discontinuous access required for filling the last column leads to increased chance of cache misses, which significantly reduces performance. The final approach strictly computes the costs row-wise and thus avoids this overhead.

The second part uses the OpenMP (Open Multi-Processing) API which GCC has support for. Both Strassen's and the block matrix multiplication algorithm were implemented and the naive algorithm was modified to its 'ikj' variant for comparison of the three algorithms. All three showed similar sequential performance for  $n = 1000$ , with the block algorithm slightly outperforming the other two ( $\approx 6.2\%$  of total runtime). However, when coupled with parallelization, the naive variant was the fastest. It is likely that the other two algorithms perform better as the input size

grows further due to memory considerations, but the current approach was selected for being the fastest on the maximum size mentioned in the problem statement.

The notation 'ikj' determines the order in which the loops appear, and thus the standard algorithm is designated as 'ijk'. Clearly, the 'ikj' variant involves swapping the order of the two inner-most loops. This variant outperforms the standard one because it is able to leverage locality of reference. In both, the inner-most loop performs the following operation:  $C_{i,j} := A_{i,k} * B_{k,j}$ . Note that when the inner-most loop is 'j', matrices  $B$  and  $C$  are indexed row-wise, which allows use of spatial locality of reference. Further, matrix  $A$  has an invariant value over the loop that can be stored in a register, which is done automatically by the compiler and is called Hoisting. This employs temporal locality of reference. As memory is often the bottleneck for modern CPU-based matrix multiplication, this small change provides a substantial improvement in performance ( $\approx 44.7\%$  reduction in total runtime). Further, CPU-based parallelization using the OpenMP API was able to improve the performance by an order of magnitude ( $\approx 87.8\%$  reduction in total runtime) on top of this.

The final approach for the filter uses OpenMP parallelization along with another loop-order change similar to the one above. As mentioned above, the total runtime contribution of this step was quite low and consequently not much time was spent exploring alternatives.

Another optimization used is to set GCC to target the AVX and AVX2 (Advanced Vector Instructions) and the FMA (Fused Multiply-Add) instruction sets. These are extensions to the instruction sets used by modern processors that allow for SIMD (Single Instruction, Multiple Data) parallelization, where the same operation is performed on multiple data points simultaneously. This change reduced the runtime by a further  $\approx 30.9\%$ .

### RESULTS AND OBSERVATIONS

Most of the runtime change results and observations have already been addressed above. Similarly, the memory drawback of the current approach with respect to the other two matrix multiplication algorithms for larger sizes of  $n$  has also been discussed.

Unfortunately, a comprehensive and accurate comparison table cannot be generated from these observations. This is because a number of changes were made in parallel while testing and the process was highly iterative. While this allowed for rapid feedback loops and quick identification of which approach worked better over the other, it had the drawback of not providing accurate quantitative data on the percentage differences between approaches. Owing to this, the values presented above are only indicative.

However, one particular observation has been replicated multiple times, and it is interesting to note that the program performs substantially better ( $\approx 26.7\%$  of total runtime) when run on WSL2 (Windows Subsystem for Linux) than on Windows, even when both are compiled with g++ with the same flags (-fopenmp -O3). The runtime on WSL2 on

Windows is around  $77\text{ms} \pm 3\text{ms}$  and on Windows with MinGW-w64 is around  $105\text{ms} \pm 3\text{ms}$ .

### FUTURE WORK

Future work on this problem should probably focus on optimizing the solution for larger values of  $n$ . It seems unlikely that further optimization could significantly reduce the runtime from the current  $\approx 77\text{ms}$ .

As mentioned earlier, this will probably mean application of Strassen's or the block matrix multiplication algorithm to overcome memory limitations. Additionally, this problem is well-suited to GPU computation for larger values of  $n$ . This is because operations like applying a filter and matrix multiplication can effectively leverage the high degree of parallelism offered by GPUs and thus are highly optimized on this hardware. Future work should look into implementation of this and to determine the cut-off for  $n$  after which the GPU overhead is sufficiently compensated by the performance increase due to parallelization.

## II. TASK 2.2: FACE DETECTION GAME

### PROBLEM STATEMENT

This task requires development of a simple game in which the user moves their face in front of a camera to control the motion of a circle. The objective of the game is to keep a ball in play by bouncing it off the circle and preventing it from reaching the bottom edge of the environment. The top, left, and right edges of the environment act as walls off which the ball bounces back. All collisions are given to be perfectly elastic and gravity is not to be considered.

### RELATED WORK

Use of face tracking in games has been explored to some extent [4], [5]. However, most of these seem to focus on face recognition or orientation of face, rather than detection and tracking of position.

Work on the actual face detection component is much more dense [6], [7], with multiple methods and libraries implementing them in Python. OpenCV contains an implementation of the cascade-classifier method discussed in [6] and other libraries implement the MTCNN approach introduced in [7].

### INITIAL ATTEMPTS

This task is fairly straightforward in terms of approach. Having developed an Atari-breakout derivative earlier, an overall design of the game environment is easy to understand. This environment includes a ball object which is responsible for all collisions, updates, and drawing itself, and it interfaces with a simple driver program that provides input from the user. For this task, the input can be represented as the face coordinates and radius in each frame. Furthermore, prior experience with OpenCV's Haarcascade-based face detection allowed for rapid development of a prototype.

The prototype also used approximations instead of the exact vector formula for the ball-face collision. The ball's  $y$ -coordinate was compared to the face's, and depending on

the range in which it fell, either or both of the  $x$  and  $y$  velocities of the ball were reversed.

### FINAL APPROACH

The initial program was designed as a prototype, and underwent two major changes - the face detection algorithm and the ball-face collision logic.

The cascade classifier worked well when the face was near the center of the frame. However, as the face moved towards the sides, its orientation would inevitably change and it would no longer remain entirely frontal. This caused the detector to miss the face in almost all frames in which it was near the edges of the environment. In some frames, multiple faces were incorrectly detected but this was largely mitigated by picking the detection with the largest area.

The final approach uses OpenCV's Deep Neural Network (DNN) module with a pre-trained Caffe model. The model is based on the Single Shot-Multibox Detector (SSD) and uses ResNet-10 architecture as its backbone. The accuracy of this model is significantly better than that of the cascade detector, and it is able to accommodate a wide range of angles. Moreover, it is able to run in real time on a CPU (Intel i7-8750H, laptop) with images of size  $1280 \times 720$ . The only drawback of this method is that accuracy drops for very large and very small faces, but that doesn't affect the performance in the game.

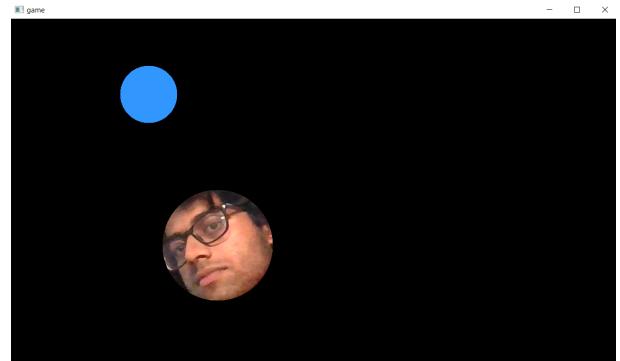


Fig. 1. Game environment with DNN-based face detection

Figure 1 shows a screenshot of the game environment. Only the face is shown using a mask, as asked in the problem statement. Further, note the successful detection despite the orientation (high values for roll and yaw angles) of the face. This demonstrates the significant improvement offered by the DNN detector as the cascade classifier fails at even smaller angles. For reference, the original program with the cascade detector is retained in the repository within the legacy folder.

The final program uses the vector equations given in [8] to compute the final velocity of the ball after collision. However, for this computation the face is assumed to be at rest at the moment of collision. The reason for this design is to prevent the ball from attaining absurdly high or low velocities, which will be the case if the head's velocity is taken into consideration. Moreover, taking the head's velocity will make the game sensitive to noise in the face detector, as

a small deviation in the face coordinates in adjacent frames can lead to a very high velocity if the time elapsed between frames is small. The final formula with the rest assumption is given in equation (1), where  $v$  is the velocity vector of the ball, and  $r_{\text{face}}$  and  $r_{\text{ball}}$  are the position vectors of the face and the ball, respectively.

$$v := v - 2 * \frac{v \cdot (r_{\text{ball}} - r_{\text{face}})}{|r_{\text{ball}} - r_{\text{face}}|^2} * (r_{\text{ball}} - r_{\text{face}}) \quad (1)$$

Finally, one aspect of the final approach is dealing with occlusion/noisy frames in which a face is not detected. During testing this situation did not occur naturally but a mechanism for handling it was added as a fail-safe anyways. The current mechanism simply reuses the last known face position, but increases the face size by a little each frame. This method is acceptable for a few consecutive frames at most where the face may not be detected due to rapid motion or some other reason.

### RESULTS AND OBSERVATIONS

As previously mentioned, the final DNN-based approach outperforms the initial cascade-based one by a large margin. Both work in real time so computational requirement of both seem to be of the same order of magnitude.

Other methods have been explored but not implemented. These include the MTCNN approach, a HOG (Histogram of Oriented Gradients) + Linear SVM detector implemented in the dlib library, and a CNN-based approach also implemented in Dlib. Comparison data available online (such as [9]) indicate that both the Dlib-CNN and MTCNN approaches are unlikely to work well in real-time due to the computational requirement. Dlib's HOG+SVM detector seems to give better performance than the cascade detector but is surpassed by the DNN model.

### FUTURE WORK

Future work could look into using Kalman Filters or similar algorithms to have a reasonable estimate of the face's position for frames where the face detection fails due to occlusion or rapid motion. Furthermore, use of GPUs can be investigated to run more complex face detection algorithms, even though the current algorithm performs more than satisfactorily.

Another thing that could be looked into is using the face's position in previous frames to estimate velocity which could be factored in to make the collisions more realistic. The reason it has not been implemented here has been detailed above. Additional mechanisms that constrain the velocity of the ball may be required in conjunction with this.

Other, more obvious changes include adding animations and improving the currently-barebones User Interface, essentially converting this project from a proof-of-concept to an actual, playable game.

### III. TASK 3.1: PATH PLANNING AND COMPUTER VISION PUZZLE

#### PROBLEM STATEMENT

This task is modelled as a puzzle and a folder is provided which contains some images and a password-protected zip folder. The task is divided into 4 levels and a hint is provided for the first level. Figures 2, 3, and 4 contain the provided images.

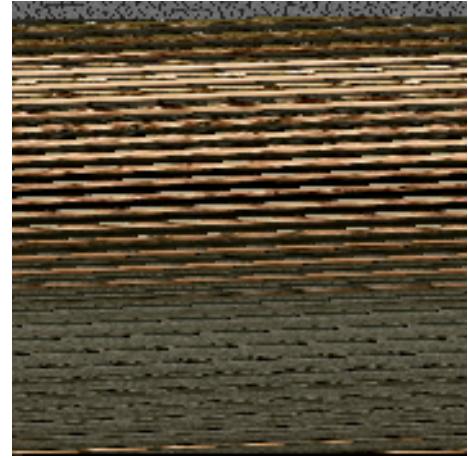


Fig. 2. Level1.png



Fig. 3. zucky\_elon.png



Fig. 4. maze\_lv3.png

### RELATED WORK

Most of the image processing techniques involved in this task are standard problems and have readily implemented solutions in libraries like OpenCV. While other methods of achieving the same end may be feasible, it is unlikely that

there are significant advantages to using those alternatives instead of OpenCV.

Further, the task requires pathfinding from a single source to a single target in a discrete maze. Several algorithms are designed to find paths in grids, and the most suitable algorithm often depends on the exact problem at hand. In this situation, A\* and its variants are likely to be the most performant algorithms due to the discrete nature of the grid and the maze-like design. Other approaches include RRT and RRT\* but they are not well-suited to this task and will be discussed further.

#### INITIAL ATTEMPTS

Levels 1, 2, and 4 were solved using a Jupyter notebook. The maze generated in level 3 was visualized in the same notebook, but a python script was written to solve the maze and retrieve the password for the zip file. The approaches for each level are fairly straightforward and thus the initial approach and the final approach are the same. The Jupyter notebook, however, has been cleaned and made presentable from its original exploratory state. All of the approaches are discussed below.

#### FINAL APPROACH

The hint provided in the problem statement mentions that both the image RGB channels and the ASCII character set use numbers from 0 to 255 to represent their data. With this hint, the grayscale pixels of the image are converted to ASCII and printed out. This reveals a congratulatory message that also provides information for the other levels.

As per the obtained instructions, the non-grayscale pixels in the first image represent an image that is to be found within the larger image ‘zucky\_elon.png’, and the x-coordinate of its top-right corner is required. To get this image, the level1 image array is sliced from the position where the text ends and reshaped to  $200 \times 150$  as instructed. The generated image is depicted in figure 5.



Fig. 5. Reshaped image for level2

The template is conveniently found within the larger image

using OpenCV’s built-in template matching algorithm. The method used is simply the first one in the documentation, TM\_CCOEFF, which computes the Correlation Coefficient between the selected ROI of the image and the template. The function returns a coefficient for each image pixel (leaving a margin at the right and bottom of the size of the template), and the pixel with the highest coefficient (1, in the ideal case) gives the top-right corner of the matched template. This corner is found to have coordinates (230, 460) and the image with the matched template is visualized in figure 6.



Fig. 6. Image with matched template highlighted

The instructions mention that the x-coordinate computed above represents the color of a monochrome maze hidden in the noisy image provided. The image is read in and the blue channel values are compared with the computed value of 230. This generates a boolean mask that selects the required pixels. The mask itself serves as a monochrome image of the maze and is rendered in figure 7. Using the axes bars drawn by matplotlib when the image is rendered in the Jupyter notebook, valid cells (20, 150) and (430, 160) are identified which serve as start and end cells. Note that the given coordinates are  $(x, y)$  while the image indices used in the code are in reverse order due to row-major indexing.

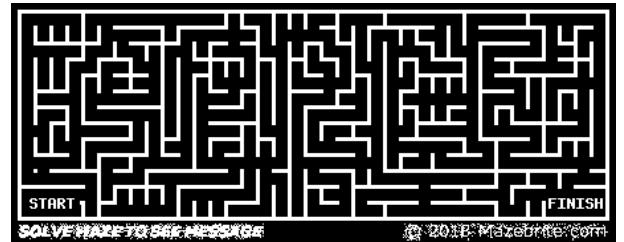


Fig. 7. Generated maze for level 3

The maze is solved using the A\* pathfinding algorithm and its variants - Dijkstra’s Algorithm and Best-First Search. Two cases are considered, one where only four - up, down, left, and right - moves are allowed, and the other in which diagonal moves are allowed as well. The solved mazes are visualized in figures 8, 9, and 10 for case 1 and figures 11, 12, and 13 for case 2. The orange cells represent explored

nodes, the blue represent the nodes on the frontier (currently in the open list/priority queue), and the purple cells represent the nodes on the generated shortest path. In both cases, the true shortest path clearly spells out the word ‘APPLE’, which is the password for the zip file.

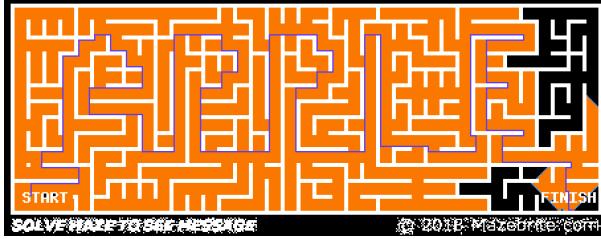


Fig. 8. Case 1 (4 moves), using Dijkstra’s Algorithm

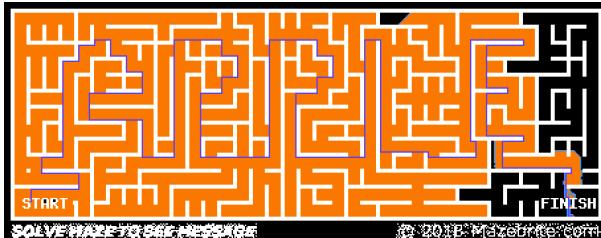


Fig. 9. Case 1 (4 moves), using A\* Algorithm

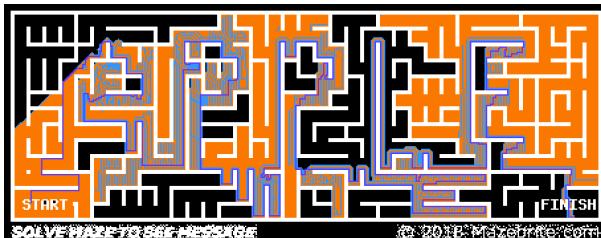


Fig. 10. Case 1 (4 moves), using Best-First Search Algorithm

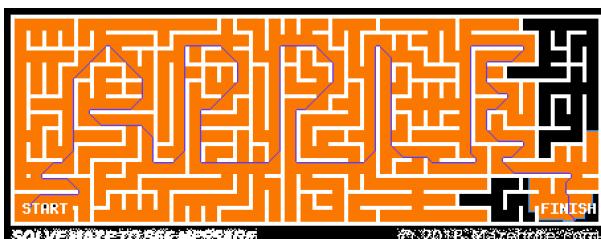


Fig. 11. Case 2 (8 moves), using Dijkstra’s Algorithm

The zip file contains a single image named `treasure_mp3.png` (shown in figure 14). The grayscale appearance of this file suggests using a similar method as in level 1, and printing out the first few characters reveal an ID3 tag which confirms that this is the right approach. The characters are written out in blocks of 10,000 to the `treasure.mp3` file which plays the **expected** song.

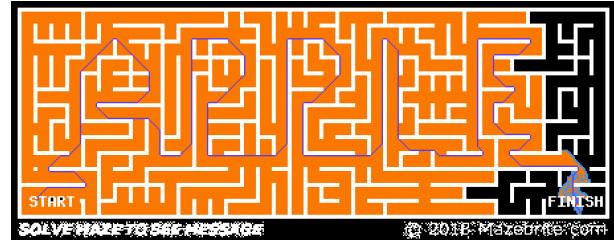


Fig. 12. Case 2 (8 moves), using A\* Algorithm

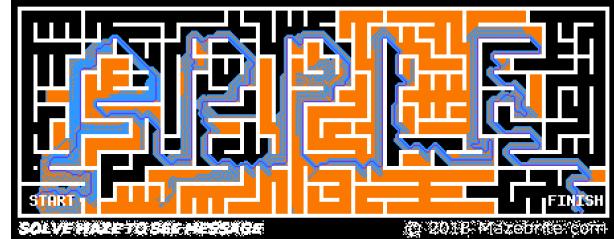


Fig. 13. Case 2 (8 moves), using Best-First Search Algorithm

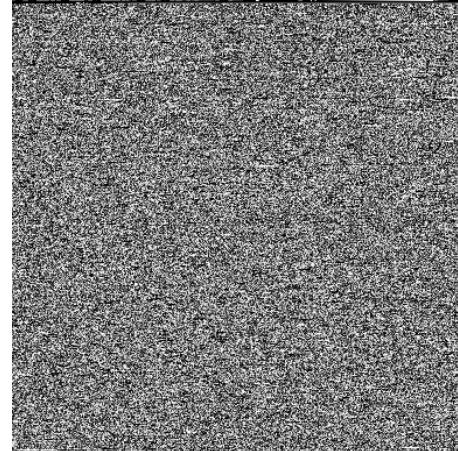


Fig. 14. `treasure_mp3.png`

## RESULTS AND OBSERVATIONS

Table I compares the number of nodes explored by the three algorithm variants in the two considered cases. The results are in line with the theoretical prediction, where A\* is able to avoid exploring as many nodes as Dijkstra due to its ‘informed’ nature. However, this effect is not very large for mazes as a lot of dead-ends are explored. Moreover, concave obstacles such as those present in mazes have a tendency to ‘trap’ A\* and have it explore unneeded nodes. Further, as expected, Best-First Search has explored significantly fewer nodes than the other two algorithms due to its greedy nature.

This reduction in number of nodes explored comes at the cost of the final path. Both Dijkstra and A\* are able to find the minimum path in both cases (1752 and 1436 nodes, respectively), while Best-First Search finds a suboptimal path (path lengths 2666 and 1708 respectively).

The most interesting observation is perhaps Table II, which compares the runtimes of the algorithms. In both cases, Dijkstra is observed to have a moderately lower runtime than

Case	Dijkstra	A*	Best-First
4 moves	46713	45086	27400
8 moves	46807	45197	22231

TABLE I  
NUMBER OF NODES EXPLORED

A\*, even though it explores more nodes. More surprisingly, the runtimes of Best-First Search is more than an order of magnitude higher than the runtimes of the other two algorithms.

All three algorithms use objects made from the same PathFinder class which is passed a heuristic class implementing a defined Heuristic Interface. This interface requires an evaluation function that takes in the coordinates of the current and the goal nodes and returns the value of the heuristic for the particular algorithm. With this design, there is no chance of the difference in runtimes being an artifact of the implementation.

Case	Dijkstra	A*	Best-First
4 moves	2.1556	2.6237	72.2432
8 moves	3.94443	4.58715	47.9632

TABLE II  
ALGORITHM RUNTIME (SECONDS)

The explanation for the difference in runtimes is provided by looking at the number of nodes in the frontier. The frontier, also called the Open List, stores the nodes that have been enqueued but have not been explored yet in the priority queue. The working of Dijkstra is such that it explores nodes closer to the start node without any bias towards the target node. This ensures that the number of nodes in the frontier at any point in time are low, or at least lower than in the case of A\*. This difference is highly exaggerated in the case of Best-First Search as it continues to enqueue nodes along the way but explores only the most promising. This leads to the open list having an enormous number of nodes as the algorithm progresses. The  $O(n \log n)$ -time operations of the min-heap based priority queue heavily punish this number. It is also possible that the size of the heap causes cache issues but this has not been profiled. These factors together are thus able to account for the enormous runtime difference.

#### FUTURE WORK

No future work is required on the puzzle itself. However, other pathfinding algorithms can be implemented as an academic pursuit and compared with the existing ones in terms of runtime, memory usage, nodes explored, etc. The reason for algorithms like RRT and RRT\* not being implemented is that they are unsuitable algorithms for this particular problem. These algorithms are extremely useful when dealing with largely open and continuous spaces. However, in a maze environment with discrete space and highly restricted paths, RRT and RRT\* are likely to perform extremely poorly.

More complicated variants of A\* like Iterative Deepening A\* (IDA\*) or Fringe Search are likely to be much better candidates than the alternative algorithms discussed above. Bidirectional A\* can also be implemented which might improve runtime at the cost of a slightly higher memory requirement. In particular, bidirectional search might be able to reduce the number of nodes in the open lists towards the end of the run. As per the observations discussed above, this is likely to have substantial performance benefit. Lastly, improving the performance of Best-First Search can be looked into by using heuristics or approaches like bidirectional search to reduce the size of the priority queue.

#### IV. TASK 4.1: TRACKING MULTIPLE OBJECTS PROBLEM STATEMENT

This task deals with localization and tracking of a drone that moves in 3D space. There are six ground stations that report the drone's relative position with respect to themselves at each time step. These measurements contain noise that is distributed according to a Gaussian. Furthermore, the first ground station is located at the origin while the positions of the other ground stations are not known. The goal is to determine the drone's trajectory over the course of its motion and also to determine the positions of the five ground stations.

Further, the input data is divided into two problem sets. Problem set 1 contains three independent CSV files that have increasing levels of noise. Problem set 2 contains a single CSV in which some data points are erroneous.

All of the four CSV files contain 10,000 rows and 18 columns. The columns are arranged in order  $(x_1, y_1, z_1, x_2, \dots, z_6)$ , where column  $x_1$  contains the x-coordinates as reported by station 1, and so on. Each row represents the measurements at a single point in time.

trajectory\_measured

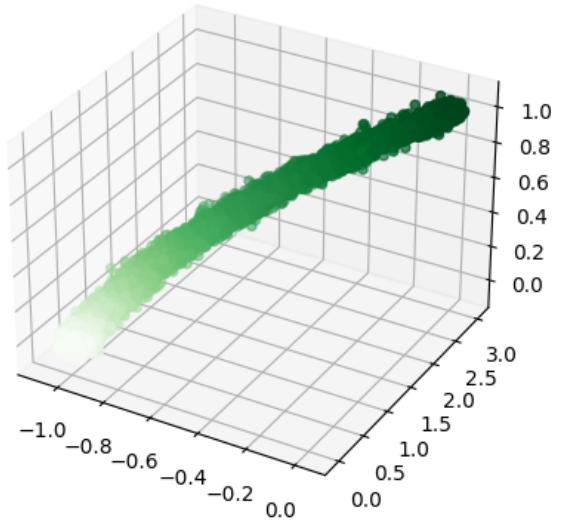


Fig. 15. Measurements given in CSV 3 with respect to station 1

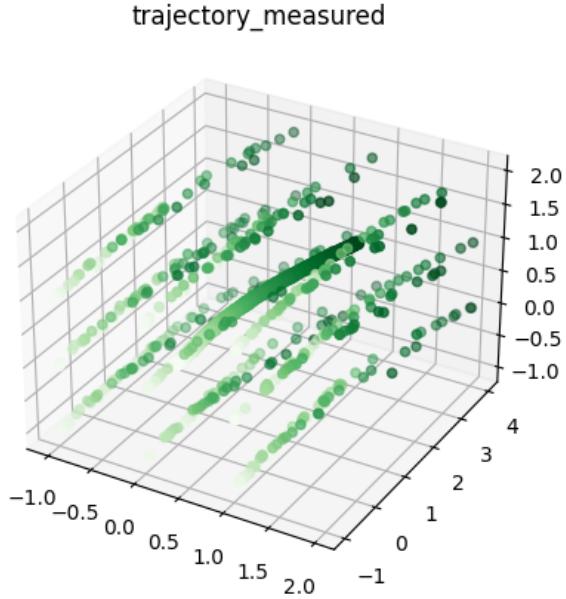


Fig. 16. Measurements given in CSV 4 with respect to station 1

Figures 15 and 16 show the trajectories given in the input data. Figure 15 is the plot for the third CSV (highest noise but no errors), and Figure 16 for the fourth (includes erroneous values). The fourth dimension needed to represent time in the plot is given by the color of each data point, which changes uniformly from white to green.

#### RELATED WORK

Filters like the variants of the Kalman Filter (Extended Kalman Filter and Unscented Kalman Filter) are widely used for purposes of tracking and localization [10]. Other commonly used techniques include the Particle Filter [11]. Most of these algorithms are targeted at different specializations of the general problem and are briefly discussed in this work.

#### INITIAL ATTEMPTS

As no model or process information is given, the first approach used a constant acceleration model for the drone's position with respect to station 1. The state tracked the position, velocity, and acceleration components of the drone, and the three position components at each time step served as the measurement values to update the filter.

Moreover, note that any station's position with respect to station 1 can be obtained by simply subtracting the drone's position with respect to that station from the drone's position with respect to station 1 at the same time step. This is shown in Equation (2), where  $x_{s_i,s_1}$  is the position of Station i with respect to Station 1 and  $x_d$  is the position of the drone in any arbitrary but fixed coordinate system. All other symbols in the equation are unambiguously determined by this notation, and the equation clearly holds for all three independent coordinates.

$$\begin{aligned} x_{s_i,s_1} &= x_{s_i} - x_{s_1} \\ &= (x_d - x_{s_1}) - (x_d - x_{s_i}) \\ &= x_{d,s_1} - x_{d,s_i} \end{aligned} \quad (2)$$

Using this, a noisy estimate of the position of all stations is obtained at each time step. A constant position model Kalman Filter can be used for this, but that simply devolves into a running mean of these estimates. For computational efficiency and convenience reasons, the running mean is directly implemented which estimates the positions of the ground stations. This method was able to achieve satisfactory output.

#### FINAL APPROACH

The constant acceleration model produced a drone trajectory that was significantly less noisy than the input but the performance wasn't satisfactory. One thing to note is that the positions of the drone with respect to the last five ground stations do not provide any meaningful information as the positions of the stations themselves are estimated using those measurements. Reusing this data will lead to the trivial equation and will not add any new information to the system. However, note that by subtracting adjacent measurements, the term that depends on the position of the stations is cancelled out, leaving only the change in position of the drone over a time step which is independent of the origin of the coordinate system. The cancellation relies on the fact that the stations are at rest in the frame of reference of station 1. This is captured in Equation (3), where  $x_{d,s_i}(t)$  represents the position of the drone with respect to Station i at time t,  $x_d(t)$  is the position of the drone at time t in any coordinate system that is at rest with respect to station 1,  $dt$  is the time step,  $\bar{v}_d(t)$  is the average velocity of the drone in the time interval  $dt$  starting at time t, and other symbols are similarly defined.

$$\begin{aligned} x_{d,s_i}(t + dt) - x_{d,s_i}(t) &= [x_d(t + dt) - x_{s_i}(t + dt)] \\ &\quad - [x_d(t) - x_{s_i}(t)] \\ &= x_d(t + dt) - x_d(t) \\ &= \bar{v}_d(t) * dt \end{aligned} \quad (3)$$

Using this property, additional information in the form of a velocity estimate can be provided to the Kalman Filter at each time step. This velocity estimate is obtained by combining the velocities as observed by the five ground stations and thus has lower variance than the position estimates. The final approach incorporates this data as well and is able to perform much better than the original by using the velocity data to improve the constant acceleration model.

The State at any time t is represented by a 9-dimensional vector:  $[x, y, z, v_x, v_y, v_z, a_x, a_y, a_z]^T$ , that is, the Cartesian components of the position, velocity, and acceleration of the drone. Corresponding to this, the initial state covariance matrix is a  $9 \times 9$  diagonal matrix, with the first 3 diagonal values being the initial position variance, next 3 the initial velocity variance and the last 3 the initial acceleration

variance. The position variance is set to unity as the absolute value is irrelevant as long as the ratio of all other variance values is constant. By setting position variance to unity, all other values are simply expressed relative to this unit.

Velocity and acceleration are initially assumed to be 0 and thus the initial variance values are set very high ( $10^9$ ). However, this value doesn't influence the calculated trajectory much as the calculated velocities are quickly able to correct the state value. The only significant impact these values have is on the second problem set. This is because the velocity mean is easily biased by outliers due to erroneous data points. This leads to high instability in the calculated trajectory near the start (first 500 readings) but eventually is smoothed out by the Filter. Interestingly, the 0 initial velocity and acceleration are good enough guesses for the given data such that setting a much lower initial variance substantially improves the performance on the second problem set without much influence on the first (marginal improvement). For this reason, the values were finally set to  $10^{-9}$  during tuning.

Other parameters like the Measurement Covariance Matrix, the State Transition Matrix, the Control Matrix, and the Measurement Transform Matrix are trivial to compute for the constant acceleration model and are sufficiently explained by taking into account the discussion above. The code has comments that further clarify this. A Process Noise Matrix is used to introduce some uncertainty at each step to prevent the model from overly relying on the prediction as it obtains more data. The matrix values are trivially derived for this model with  $dt = 1$ . Further, the matrix requires a coefficient multiplier which is set to  $10^{-20}$  during tuning. This extremely low value confirms that the velocity estimates are quite accurate and the constant acceleration model is a good fit for this problem.

The implementation of the Kalman Filter is standard, but uses the Joseph Form of the Process Covariance Update Equation instead of the simplified equation. This is because the simplified equation is numerically unstable, and floating point errors during the identity matrix subtraction can lead to certain non-negative variables having negative values. This negative value might seep into the Kalman Gain calculation in future iterations and cause the Filter to diverge. Testing shows that the maximum observed difference between the equations is of the order  $10^{-22}$  (maximum  $\approx 4.2 \times 10^{-22}$ ), and while it has no observable effect on the accuracy in this problem, the Joseph Form is retained to make the Filter generalize to other problems better. The Object-Oriented design of the Filter is also with the aim of generalizing it to other problems.

While the problem statement mentions that bonus points will be awarded for more complicated algorithms like the Particle Filter, a conscious decision was made to continue with the Kalman Filter. This is because the Kalman Filter is optimal when the following two assumptions hold: a) All the state transitions are linear and b) All noises are distributed according to a Gaussian function. Both of these assumptions hold in this situation and the Particle Filter would thus offer no improvement in performance while

increasing computational requirement.

## RESULTS AND OBSERVATIONS

`trajectory_updated`

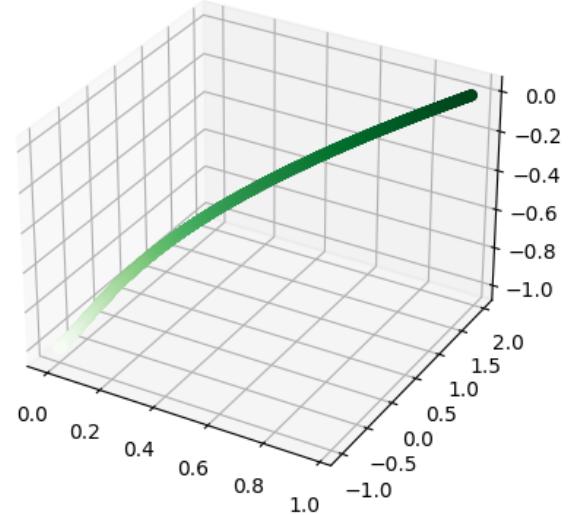


Fig. 17. Updated trajectory for CSV 1

`trajectory_updated`

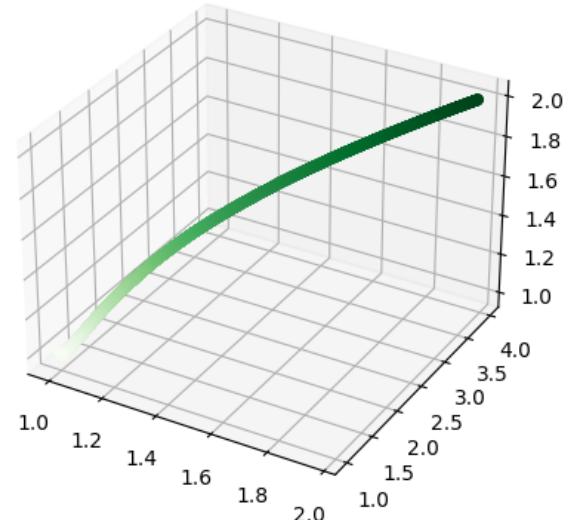


Fig. 18. Updated trajectory for CSV 2

Figures 17, 18, 19, and 20 show the updated trajectories for all four CSV files. The sheer number of data points prevents the precision of the output from being truly appreciated. To rectify this, figures 21 and 22 show the last 200 points in the input and output trajectories for CSV 3. The last two figures are able to demonstrate the precision of the Filter.

trajectory\_updated

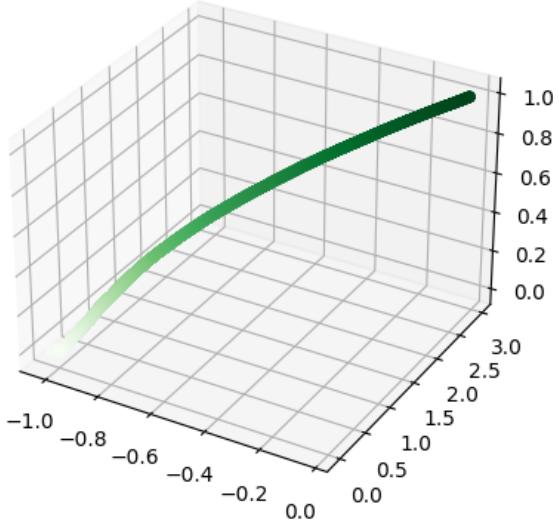


Fig. 19. Updated trajectory for CSV 3

partial\_trajectory\_measured

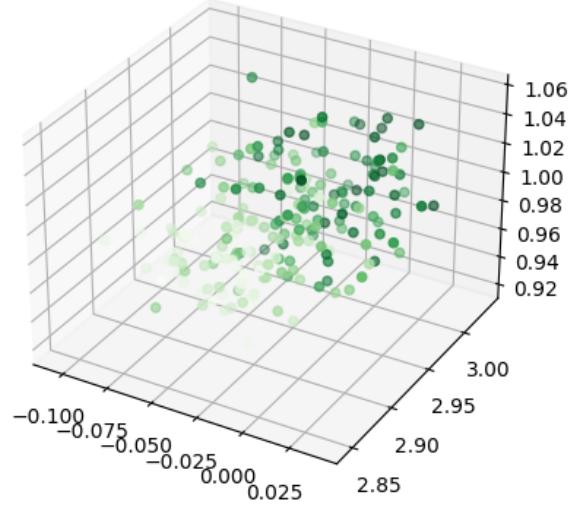


Fig. 21. Partial measured trajectory for CSV 3

trajectory\_updated

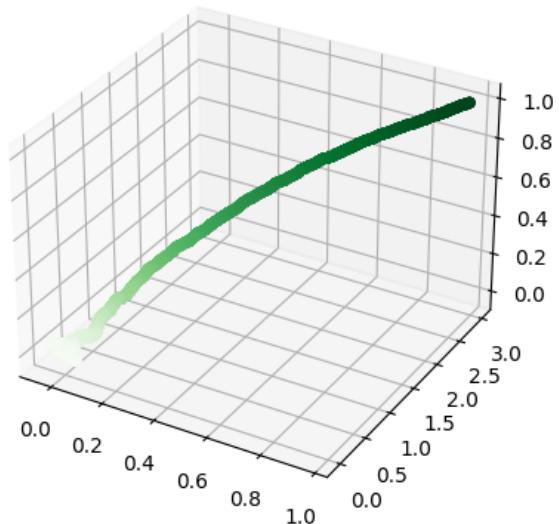


Fig. 20. Updated trajectory for CSV 4

partial\_trajectory\_updated

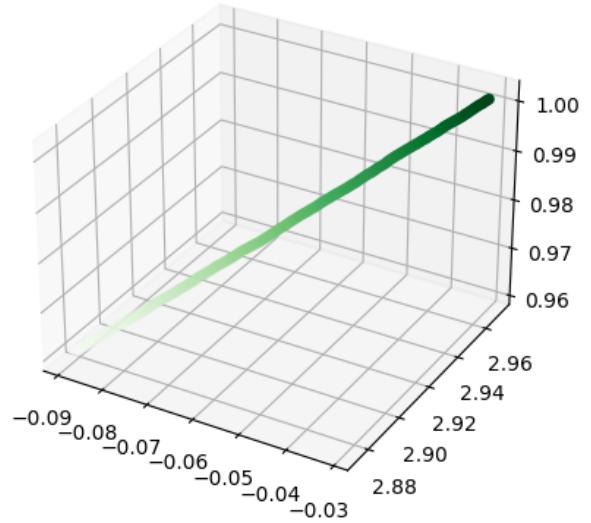


Fig. 22. Partial updated trajectory for CSV 3

The positions of the ground stations are computed independently for each CSV file, and are extremely close to integers  $\times 10^{-1}$ . The maximum deviation from these assumed-to-be-correct integral values is  $2.7 \times 10^{-3}$ , which corresponds to a relative error of  $\approx 0.54\%$ . The values are omitted from this document for brevity and can be conveniently found in the output folder of the code repository corresponding to this task.

#### FUTURE WORK

The performance for the first problem set is more than satisfactory and aside from further tuning, it seems unlikely

that it can be improved substantially. Moreover, as mentioned earlier, more advanced techniques such as the Particle Filter are extremely unlikely to yield better results due to the nature of this problem. The Extended and Unscented variants of the Kalman Filter are similarly not expected to perform better as they are designed to handle systems which do not have all state transitions as linear.

Future work on the second problem set should probably focus on detecting and removing outliers, and perhaps switching to some combination of median and mean for estimating the velocity from the five stations. Both of these techniques essentially engage with the root cause for the performance

on the second problem set not being as good, and thus are likely to yield optimal performance if implemented correctly,

In a real-world scenario, using data from other sources such as the drone itself can be very useful in further improving accuracy. In the absence of this, computing estimates for the positions of the other five stations pre-flight can allow full use of the data provided by the stations.

## V. TASK 4.2: 3D TIC TAC TOE AGENT

### PROBLEM STATEMENT

This task is divided into two parts. The first part requires implementation of a 2D Tic Tac Toe agent that uses the minimax algorithm. The second part deals with the 3D extension of the game, where the grid dimensions are  $3 \times 3 \times 3$ . This significantly increases the state space making the exhaustive search prohibitively expensive. As per the problem statement, this part requires optimising the minimax algorithm or using alternate techniques such as Q-Value Learning or Genetic Algorithms,

Moreover, an implementation of the 3D variant is provided in the form of an OpenAI Gym environment and a template using the same is also provided.

### RELATED WORK

Minimax algorithms and its variants like negamax have been widely used for turn-based games like Go, Chess, Backgammon, etc. More recently, however, Reinforcement Learning techniques have been used to overcome the computational requirement of exhaustive search algorithms which require several heuristics to be feasible on games like Chess. Reinforcement Learning is an extremely broad field with several different approaches and algorithms, but generally employs machine learning algorithms to allow an agent to learn strategies by interacting with an environment. For turn-based games such as 3D Tic Tac Toe, algorithms like Deep Q Learning and the Monte Carlo Search Tree algorithm have been used with much success.

### INITIAL ATTEMPTS

The initial attempt for the first part is unchanged in the final submission. It uses the negamax variant of the minimax algorithm (explained below) along with memoization. Alpha-Beta pruning was implemented as well, but it reduced the efficacy of memoization and the total computation time over the entire game increased. For this reason, the final approach reverted to a pure-memoization approach.

Some testing was done by adapting the algorithm for the second part, and it could compute about 1,000 states per second. This was much lower than expected, and can partially be attributed to factors like poor design of the environment (using pure Python instead of numpy which would speed up a lot of operations and reduce the interpreter overhead), and general inefficiency of Python due to its interpreted nature. If implemented properly with optimizations in a language like C++, this method can be feasible especially with a precomputation period allowed. However, this approach was entirely dropped for a Python implementation due to 1,000

states/second being orders of magnitude lower than what would be required.

### FINAL APPROACH

As mentioned previously, the solution for the first part uses the Negamax variant of the minimax algorithm. The Negamax variant requires that the game be zero-sum, and the algorithms are functionally identical under this assumption. The difference is a small implementation change which makes negamax slightly more convenient. More specifically, it exploits the property that  $\min(a, b) = -\max(-a, -b) \forall a, b \in \mathbb{R}$ . Thus, instead of having a minimizing agent and a maximising agent, it suffices to have both agents trying to maximize their own reward. This is where the zero-sum property allows easy conversion between rewards of player 1 and player 2 by multiplying by -1. In practice, this means that a simple max can be taken at each step by multiplying the evaluations of all children by -1.

The memoization in the 2D case uses a function that maps states uniquely to integers that serve as indices in the memo table. This uses the fact that the representation of the board is a  $3 \times 3$  grid of integers 0, 1, and 2. Note that a string of 9 digits is obtained by flattening this grid, where each digit takes values 0, 1, and 2. This suggests an extremely convenient one-one mapping between states and ternary numbers, which are simply converted to decimal for the indices. In practice, the implementation avoids the string conversion and requires only multiplications and additions. Other methods like making a string out of the board and using a dictionary might be more convenient but are computationally more expensive due to hashing and other operations that are required. However, the difference in runtime is likely to be negligible and as such has not been benchmarked.

As mentioned earlier, the minimax approach for the 3D game is impractical. The final approach uses Deep Q Learning, but has not been entirely implemented due to time constraints. The approach and several implementation details are discussed here, along with some preliminaries.

Q-Learning is a model-free reinforcement learning algorithm which computes Q-Values for (state, action) pairs. The Q-Value, or Quality, indicates how beneficial a particular action is for the agent at a given state. More formally, the Q-value of a (state, action) pair is the overall expected reward if the agent performs the action at the given state. Due to the definition of Q-value as the overall reward rather than the immediate reward, it is trivial to see that taking the action with maximum Q-value at each state is an optimal policy.

The problem is now reduced to computing and storing these Q-values. Vanilla Q-Value Learning does this by randomly interacting with the environment until it encounters a state with non-zero reward. It then uses this reward to update the Q-values estimates (guesses, initially) for the states it has traversed by iterating over them backwards. The backwards iteration is done because the reward is often discounted as it moves back in time. This discounting allows an agent to differentiate between immediate and deferred rewards and a trade-off can be made by varying the discounting factor

(often represented as  $\gamma$ ). These values are stored in a Q-value table and is then used to make decisions.

Due to the explicit table representation, the agent needs to have previously explored a state at least once to be able to make meaningful decisions for that state. This is a huge drawback and makes the approach infeasible for this task. Deep Q-Learning counters this by using a Neural Network to approximate Q-values without needing a lookup table. Thus, a well-trained and -designed network can work well even when trained on a tiny fraction of the state and action space.

Deep Q-Learning is not without demerits. An issue that this approach faces has to do with how the target Q-values are generated for training the network. The formula that is used for this is derived using Bellman's Principle of Optimality and is essentially the sum of the immediate reward and the discounted Q-value of the state achieved after performing an action. Realize that the Q-value of the next state is also computed using the neural network being trained, which means that updating the network causes the target values to update as well. This leads to highly unstable learning and can cause the algorithm to diverge. Moreover, learning from continuously consecutive experiences is inefficient due to strong correlations between the states. In fact, this can lead to the network overfitting a subset of states.

The first issue is commonly mitigated using a separate target network that generates the target values for training. Periodically, the weights of the predictor network are copied over the target network. This immensely improves learning stability and prevents the network from diverging. The second issue is combated using a technique called Experience Replay. The idea behind this technique is to accumulate experiences and then to learn in batches by sampling from this memory. The random sampling is able to overcome the problem of overfitting if done over a large enough set of experiences. Further, this has additional advantages such as allowing an agent to learn from a single experience multiple times without having to revisit and compute the state. It also allows offline learning which can be done in batches to speed up computation. The disadvantage of this method is the increased memory footprint.

A few other concepts are involved in this approach such as exploration-exploitation. This is a decision that the agent needs to make to determine whether it should randomly take actions in order to explore the environment better and gain more information about it or if it should exploit the previously gained information and act to maximise its reward. A commonly used policy for this is the epsilon-greedy policy, which uses a parameter  $\epsilon$  which represents the probability of the agent choosing to explore by taking a random action from the action space. Often, epsilon is set to a high value at the start and is reduced as the agent gains more information about the environment. This variant is called the decaying epsilon-greedy strategy.

The Neural Network can be conveniently implemented using any of the several machine learning frameworks available. For this task, the implementation uses the Keras API with TensorFlow 2 as the backend. The agent itself is

implemented as an object which uses this neural network model. The weights for the model can be read in from a file or trained and written to disk. An epsilon-greedy strategy is implemented for decision making during training and a greedy strategy for decision making during gameplay.

The memory required for Experience Replay is implemented as a helper class which provides convenience functions to store and sample experiences. Each experience is represented as a 4-tuple which contains the initial state, action, final state, and reward. The implementation of this has been optimized by representing the memory as a tuple of numpy arrays rather than a Python list of tuples. This allows operations to be performed in batches.

Most of the mentioned features are entirely implemented, with the exception of the training pipeline which could not be completed within the time constraint.

## RESULTS AND OBSERVATIONS

The negamax implementation in the 2D case is sufficiently fast and is able to compute evaluations for the entire search space in  $\approx 0.54s$  as the first player and  $\approx 0.16s$  as the second player. After this initial computation, each move is selected in less than 1ms. The computed evaluations can also be stored on disk and simply read-in during runtime, but that has not been implemented.

Unfortunately, the Deep Q Learning agent for the 3D case has not been completely implemented and thus no observations are available.

What is interesting to note is that the problem statement mentions that Q-Value Learning can be used for this problem. However, as discussed above, Q-Value Learning is expected to be computationally much more taxing than even a naive minimax agent for this problem, and even then isn't guaranteed to perform optimally. Q-Value Learning performs well in smaller, non-deterministic environments where standard minimax does not work optimally as it assumes a deterministic environment.

## FUTURE WORK

Future work on the 2D part, if any, will probably be academic or exploratory in nature. In practice, the tiny state space of the game ensures that even the most naive approaches are viable and thus the approach presented here is more than sufficient for any practical requirement. However, this also makes the game a convenient playground to test and debug other approaches and algorithms.

Future work on the 3D portion would undoubtedly involve completion of the implementation of the Deep Q-Learning agent. Tuning parameters such as discount factor, learning rate, batch size, etc. and optimizing the network architecture is also likely to be required.

Aside from this, algorithms such as the Monte Carlo Search Tree or model-based Reinforcement Learning algorithms can be implemented for this problem. Another possible approach is to use the genetic algorithm.

## CONCLUSION

The tasks were quite enjoyable and provided a welcome learning experience. While not being particularly difficult, each task definitely required a good mix of reading, thinking, and programming. The exception to the trend in difficulty, at least for myself, was the second part of the final task which required a lot of reading on Reinforcement Learning, not to mention the time taken in implementation of the Deep Q Learning agent.

What stands out, however, is that all of the concepts involved in the tasks are highly practical and have applications across several fields as mentioned in the abstract. Owing to this, the work presented here is not only useful with respect to ARK, but in general as well.

## REFERENCES

- [1] Gunnels, John, et al, ‘A Family of High-Performance Matrix Multiplication Algorithms’, Applied Parallel Computing, State of the Art in Scientific Computing, 7th International Workshop, PARA 2004, Lyngby, Denmark, June 20-23, 2004, Revised Selected Papers, 3732, 256–265, 2004.
- [2] Hemeida, A.M., et al, ‘Optimizing matrix-matrix multiplication on intel’s advanced vector extensions multicore processor’, Ain Shams Engineering Journal, Volume 11, Issue 4, Pages 1179-1190, 2020,
- [3] Huang, Zhibin, et al, ‘GPU computing performance analysis on matrix multiplication’, The Journal of Engineering, 2019.
- [4] Sko, Torben & Gardner, Henry J, ‘Head Tracking in First-Person Games: Interaction Using a Web-Camera’, Human-Computer Interaction – INTERACT 2009, 342–355, 2009.
- [5] Demir, Ugur & Ghaleb, Esam, ‘A Face Recognition Based Multiplayer Mobile Game Application’, Artificial Intelligence Applications and Innovations, 214–223, 2014.
- [6] Viola, Paul & Jones, Michael, “Rapid Object Detection using a Boosted Cascade of Simple Features”, IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2001.
- [7] Zhang, Kaipeng et al, “Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks”, IEEE Signal Processing Letters 23.10: 1499–1503, 2016.
- [8] Christian, ‘Two-Dimensional Collisions’, Scipython Blog, 2019, <https://scipython.com/blog/two-dimensional-collisions/>.
- [9] Esler, Tim, ‘Comparison of face detection packages’, Kaggle Forum, 2020, <https://www.kaggle.com/timesler/comparison-of-face-detection-packages>.
- [10] I. Ullah, et al, ‘A Localization Based on Unscented Kalman Filter and Particle Filter Localization Algorithms’, IEEE Access, vol. 8, pp. 2233-2246, 2020.
- [11] B. Allik, ‘Particle filter for target localization and tracking leveraging lack of measurement’, 2017 American Control Conference (ACC), pp. 1592-1597, 2017.