Aerial Robotics Kharagpur Documentation

Ankit Ranjan

Abstract: The first task was to optimize the program to allow it to run on matrices of sizes up to 1000*1000. The time complexity of the functions FindMinCostA and FindMaxCostB was of the order 2^n.

The second task includes designing a game that detects my face. I had to track the movement of my face assuming my face to be a perfect circle. I used Haar Cascades classifier for face detection and used simple mathematics and laws of Physics to design the game.

The third task required the use of a computer vision module to solve a puzzle, and I used openCV-Python for the same. As part of path-finding algorithm, I used dijkstra and A* algorithm with various heuristic functions on the maze and compared their runtime.

INTRODUCTION

I had to optimize the given cpp code. The code involved two functions to be optimized, FindMinCostA() and FindMaxCostB(). The initial time complexity of the functions was of the order 2^n, where n*n was the order of the matrix to be used in the function.

In the second task, I had to code a face development game, using any computer vision module. We were allowed to use library function.

As part of the third task, we had to solve a picture-based puzzle. There were multiple levels and images to be solved. The final maze had to be solved using some path-finding algorithm. The solution to the puzzle gave the password to a protected zip file, which contained a image which finally had to be converted to audio.

TASK 1: OPTIMIZE ME

PROBLEM STATEMENT

As part of this task, I was allotted a cpp code file, which I had to optimise. The program defined two functions FindMinCostA() and FindMaxCostB().

The function FindMinCostA() gives the minimum possible cost of the path to reach the index (n-1,n-1) from any given index (i,j) in the matrix costMatrixA of size n*n, while the function

FindMaxCostB gives the maximum cost for the same in the costMatrixB of the size n*n.

The initial time complexity of both the functions was given by:

$$T = O(2^n)$$

where n is the size of the matrix on which I run the functions.

For a matrix of size 1000, T is of the order $O(2^{1000})$ which is tremendously large. So the program can't work for such a large n.

After that, I had to fill the matrix ProductMat, using matrix multiplication of the values of the costs returned by the functions FindMinCostA() and FindMaxCostB().

In the last part of the task, I had to optimize a filter application on the matrix ProductMat.The dimension of the filter given to me was `4` x `n`.

RELATED WORK

I tried to learn about different methods of optimization using online resources like StackOverflow, youtube, and some other platforms.

In the process of completing this task, I learnt and tried dynamic programming, memorization algorithm for recursion.

I also tried to learn parallel programming.

INITIAL ATTEMPTS

Initially, I tried to use dynamic programming to optimize the program using a bottom-up recursion approach. I implemented dynamic programming using memorization.

But for already filled values of the arrays, the approach didn't give the output as required which I found by dry running the code. Hence I changed my approach.

FINAL APPROACH

In my initial attempt, there was one implementation issue. Whenever the functions, FindMinCostA() and/or FinaMaxCostB() was called, some indices of the memorization matrix were

filled with values which were visited while travelling from (n-1,n-1) to that index, where n*n represents the size of the matrix.

I tried out dry run to find out the actual problem. The solution to that problem which I implemented was to follow a top-down approach, i.e. instead of dynamically programming the recursive method, I traversed tha array *from* the index of which I had to calculate the path, say (i,j) { 0 ≤ i , j < n }, *to* the final index, i.e. (n-1,n-1).

This allowed me to fill the entire memorisation table in a single function call. Therefore, I called the function once, after initialisation of the cost matrices CostMatrixA and CostMatrixB, in the *main()* function for the (0,0) index.

The calling of the functions FindMinCostA() and FindMaxCostB() filled their memorization table with values of the minimum and maximum cost of traversals, respectively, corresponding to every index.

Thus, now I had simply two matrices, i.e. memorization tables with the corresponding cost values in each cell of the matrix.
To return the value of cost required of any index, I could simply return the value of that index in the corresponding memorization matrix.

I figured out that ProductMat matrix can now be easily constructed by matrix multiplication of the two memorisation matrix, say memA and memB respectively.

I optimised the codes in the above stated manner, and now I didn't need to call both the functions for every iteration in ProductMat filling, rather I can directly use the values in memA and memB.

I tried a cache-friendly method for multiplication of memA and memB, which gave me significant improvement in runtime.

In Spite of these, since matrix multiplication has a time complexity of order $O(n^3)$, when the size of the matrix is large, it increases the overall runtime of the program.

The overall runtime is governed by the matrix multiplication to fill the matrix productMat, and the filter application on the matrix isn't consuming much time and resources, I preferred not to modify that section of code as the end result of modification won't be significantly observable.

RESULTS AND OBSERVATIONS

Algorithm	Time taken(n= 500)	Time taken(n= 1000)
Without optimising	NA	NA
After	2.1 s	16s

optimising, without using cache-friendly method		
Using cache-friendly method	3.5s	11.43s

Using a cache-friendly method gives some improvement in runtime for large array size, but for smaller size of array, due to more calculation involved, it increases runtime.

FUTURE WORK

I am currently exploring the domain of parallel programming and computing which will help me to further optimise the matrix multiplication and thus help reduce the runtime of the entire program.

The cache-friendly multiplication isn't a good technique for matrices of sizes around 500. So, I will also try to plug that loophole.

Task 2: Face Detection Game

PROBLEM STATEMENT

The objective of this task was to create a webcam-based game which tracks the face of the person standing in front of it, using any image processing library. I could have assumed the face to be a perfect circle.

There would be a ball which would bounce off the edges or the boundaries of the game window, and should be prevented from touching the floor, i.e the bottom of the game window by appropriate movement of the face. If the ball falls on the ground, the game is over, and the program stops after printing the same message('Game Over') on the window.

I had to ignore gravity and consider uniform motion and elastic collision of the balls and the moving ball and the edges of the game window.

The use of library functions was allowed.

RELATED WORK

I explored the cascade classifiers in openCV, which helps in object detection. Further, Mathematical equations involved in the logic creation of game were derived. I explored CNN too.

INITIAL ATTEMPTS

Initially, I used the cascade classifier for face detection, derived and applied the mathematical equations for collision and uniform motion.

FINAL APPROACH

I used cv.CascadeClassifier() function for the detection of the face of the person. I implemented the logic for the different collisions following the laws of physics(uniform motion and elastic collision).

Haar Cascade classifier is a machine learning-based method for object detection, trained with a large number of training samples and available to be used by anyone wanting to develop similar applications.

While the initial attempt did face recognition well, using cascade-classifiers, there were some glitches, and the game was not running smoothly when the collision of the ball took place. The reason being sudden increase in the value of velocity and sudden change in position of the free ball after collision.

Therefore some periodic updates in the value of position and velocity were applied for smooth running of the game.

Variable definitions :

```
# Resolution of game window
h = 700
w = 1364

# Ball radii
Rball = 100  # ideal is 100
Robstacle = 100 # ideal is 90

# initial parameters of 1st ball
x1 = 350
y1 = Rball + 1
```

```
# initial parameters of 2nd ball
x2 = h - Robstacle
y2 = w - Robstacle - 1

# Ball speed
vx1 = -5.0  # speed in vertical
direction
vy1 = 5.0  # speed in Horizontal
direction
vi = v1 =   5.0 * math.sqrt(2)

# Collision condition
Rtotal = Rball + Robstacle
```

(x1,y1) = coordinates of the centre of the moving ball

(x2,y2) = coordinates of the centre of the circle representing the face

(vx1,vy1) = speed of the moving ball in (x,y) directions

Conditions and equations for boundaries and ball collisions:

```
 # check for boundries collisions
for ball 1
    if x1 <= Rball :
       vx1 = -vx1
       x1 = Rball
    if y1 <=  Rball:
       vy1 = -vy1
       y1 = Rball
    if x1 >= (h - Rball) :
#changed here
       font =
cv.FONT_HERSHEY_SIMPLEX
```

```
        cv.putText(img,' Game
Over',(382,350), font,
3,(0,0,0),2,cv.LINE_AA)
        cv.imshow('test',img)
        cv.waitKey(0)
        break
    if y1 >=   (w - Rball):
        vy1 = -vy1
        y1 =   (w - Rball)
```

Conditions and equations for ball to ball collision:

```
elif math.sqrt((x1 - x2)*(x1 - x2)
+ (y1 - y2)*(y1 - y2)) <= (Rtotal
+ 1):
        v1 = math.sqrt(vx1 * vx1 +
vy1 * vy1)

        d = math.sqrt((x1 - x2)*(x1
- x2) + (y1 - y2)*(y1 - y2))

        Theta =
math.acos((((x2-x1)*vx1)+((y2-y1)*
vy1))/(d*v1))
        alpha =
math.acos((x2-x1)/d)

        vx1 = -((v1 *
math.cos(Theta) * math.cos(alpha))
+ (v1 * math.sin(Theta) *
math.sin(alpha)))
        vy1 = ((v1 *
math.sin(Theta) * math.cos(alpha))
- (v1 * math.cos(Theta) *
math.sin(alpha)) )
```

Optimisation of value of velocities to overcome glitches due to integral pixels:

```
 # optimization of value of
velocities because of pixel
problem
        if 0.3 <= vx1 < 1:
            vx1 = 1.0
        elif -0.3 <= vx1 < 0.3 :
            vx1 = 0.0
        elif -1 <= vx1 < -0.3:
            vx1 = -1.0

        if 0.3 <= vy1 < 1:
            vy1 = 1.0
        elif -0.3 <= vy1 < 0.3 :
            vy1 = 0.0
        elif -1 <= vy1 < -0.3:
            vy1 = -1.0
```

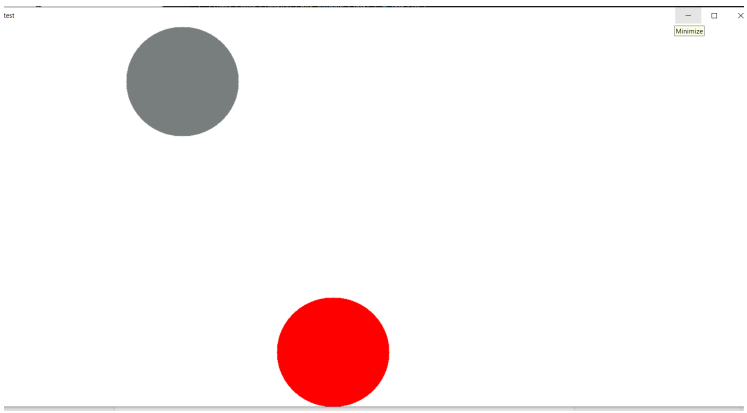

Fig 1. The face detection game window.

RESULTS AND OBSERVATIONS

The Cascade Classifier works quite fine for face detection, but is quite slow. The results are satisfactory. The updation of values of velocity and position were required to prevent sudden change in position visible in the game window.

FUTURE WORK

While the logic and mathematics behind the ball movement and collision is working satisfactorily, improvement can be made in the face detection algorithm, like using neural networks models such as CNN.

I am currently working on CNN as I have some experience in various machine learning models including neural networks, and I will try to implement it as the face detection algorithm.

TASK 3.1 : PATH PLANNING AND COMPUTER VISION PUZZLE

PROBLEM STATEMENT

The task consisted of a few images, solving which I could go to next levels, and could finally find the password to a password-protected zip file.

The first level consisted of the image ‘Level1.png’. I had to read the image to a variable and I saved the np.uint8 image matrix to the output file. Then I had to take the first values of the RGB values of the pixels from the file for which all the values were equal, and using those values as ASCII codes, I had to decode it to reach to the further levels.

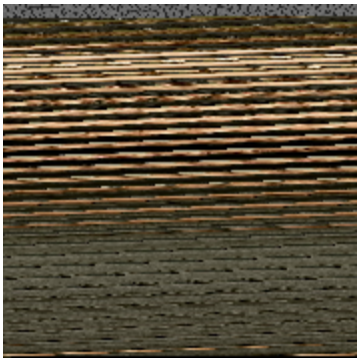

Fig.2 Level1.png

The pixel values in the file after the ASCII code hints, were the values for a (200,150,3) image, which needed to be found.

Then, that image had to be template-matched with a larger image, ‘Zucky_elon.png’ and the top left corner of the matched rectangle had to be used to find a maze from the provided, noisy image. That maze then needed to be solved with different path-finding algorithms and the solution of the maze contained the password of the zip file. That zip file had an image that needed to be converted to audio, which was the actual treasure.

Fig.3 zucky_elon.png

Fig.4 maze_lv3.png

RELATED WORK

I had some prior experience with path-finding algorithms, like dijkstra, A* etc. I tried to find out the best-suited heuristic function for the given maze.

INITIAL ATTEMPTS

I tried to decode the first image. Initially I tried all the values of the pixel and converted it to English alphabets, which didn't give any meaningful information, but gave the hint to proceed in the right direction. After a few trials, I could find the original message.

FINAL APPROACH

I used openCV-python and loaded the image to a variable and saved the image matrix to a file. Then I converted the ASCII code to alphabets, got the congratulatory message along with the hints.

Following the instructions, I used the remaining pixel values in the file to find the first hidden image, which was Zuckerberg's image. To find that image, it was instructed that the required resolution is (200,150,3). So, I did the required maths, to arrange the remaining pixels in the same resolution.

Fig.5 Solution to level1

After that, I used openCV template matching function to find exactly that part of the image in the larger image, 'zucky_elon.png'.

Fig.6 Template matching

I got the required top left coordinate of the rectangle as 230.

This 230 could represent any of the B,G or R value of the pixel in the 'maze_lv3.png'.

So I tried all possible combinations to extract pixels having that values, and red and blue gave the maze in nice colour.

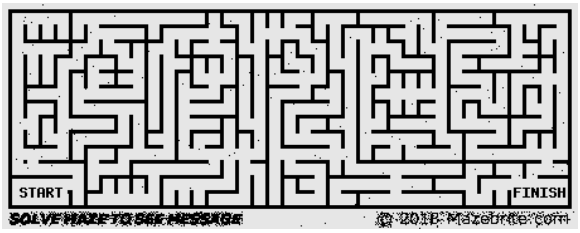

Fig.7 Maze solved

I then applied various path-finding method to solve the maze, and found the hidden password as ‘APPLE’. I used that password to unlock the zip file, and found that it contained an image ‘treasure_mp3.png’.

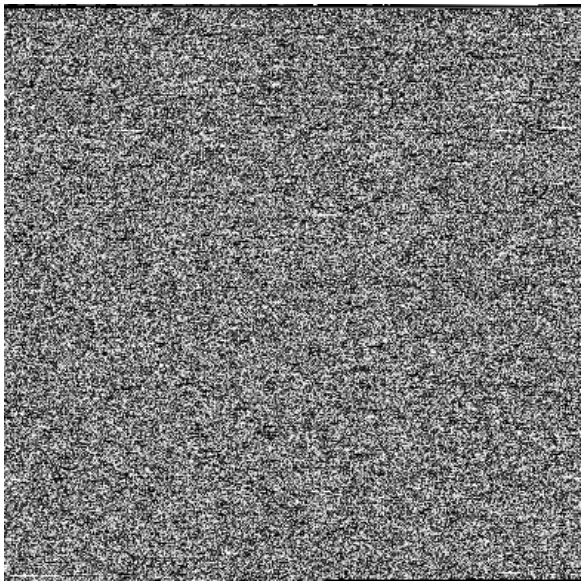

Fig.8 Template_mp3.png

I tried to convert that image into mp3 audio.

RESULTS AND OBSERVATIONS

I used different path colours for better visualisation of the path found.

Algorithm	Time taken(sec)
Dijkstra	462.798
A star(Non admissible heuristic)	375.01
A star(Euclidean Distance)	452.52

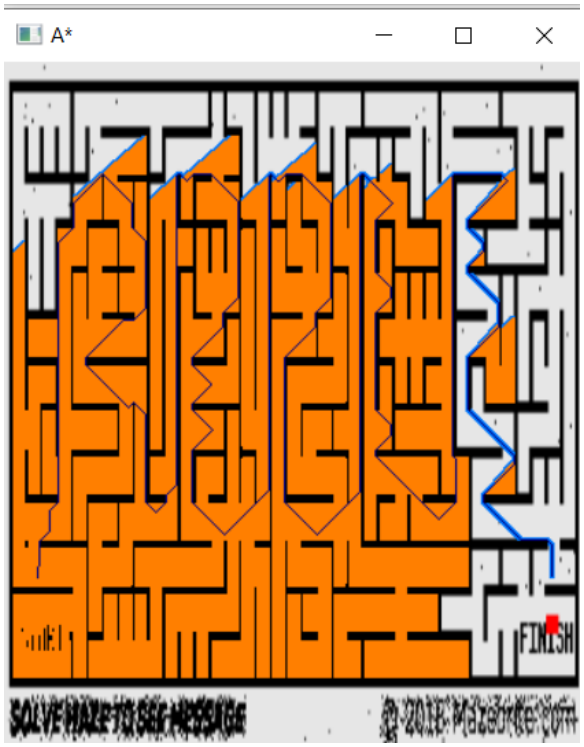

Fig. 9 Path found using Non-admissible heuristic in A star algorithm

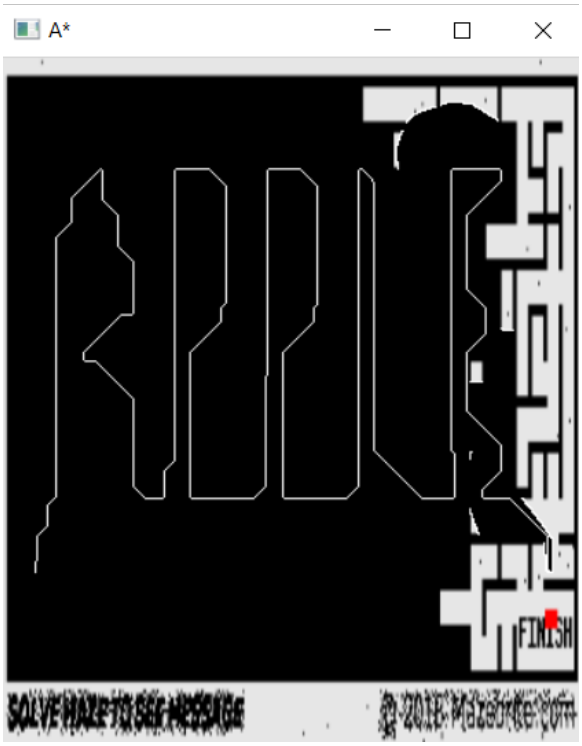

Fig. 10 Path found using Euclidean distance as heuristic function in A star.

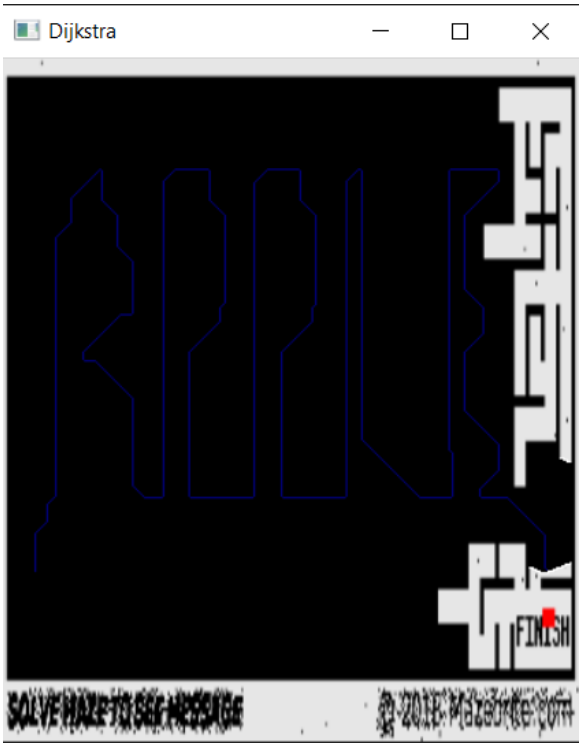

Fig. 11 Path found using Dijkstra Algorithm

FUTURE WORK

While most of the task was straightforward and wasn't much of an algorithmic approach, path finding algorithms. I didn't use RRT and RRT* as asked in the task, these are not suited for solving maze due to the degree of obstacles. I will further try to optimize the heuristic and use better path-finding algorithms.

CONCLUSION

The first task was to optimize a cpp program that contained functions and matrices. Initially the program couldn't run for a higher - sized array, but after I optimised it, there was a massive improvement in runtime. I tried and implemented some programming techniques and algorithms new to me, like dynamic programming, memorization etc.

Since to obtain efficiency in program and reduction in runtime is vital for the program to be deployed or installed on any autonomous robot system, hence the above problem and its solution would be helpful to ARK.

Second task was creating a face detection game. The cascade classifiers are of great use for face detection. Since object detection algorithm and its efficiency is detrimental to any self

driving or flying object, hence an optimal solution to the problem is of great importance.

The third task was the most interesting of all. There was an entire picture-based puzzle to solve to reach the treasure! Great deal of image processing and other computer vision applications were involved in that. Further path-planning is extremely important to find optimal path to traverse in a locale, avoiding obstacles.

REFERENCES

[1] Geeks for Geeks
[2] Stack Overflow
[3] Springer
[4] Science Direct
[5] YouTube
[6] Coursera
[7] Udemy