Task Round Documentation (ARK)

Satwik Chappidi *

***Abstract*— This document encapsulates my attempts and approaches for the various tasks of the task round for my selection into Aerial Robotics Kharagpur (ARK)**

Over the course of a month, I have made an agent that plays the game of Tic-Tac-Toe using the minmax algorithm. I have also made a simulator for a billiards game and a basic game controlled with the movement of the head. I also tried implementing Rapidly-exploring Random Tree (RRT) to guide a unit to a weapon in a game.

These tasks have led me to learn more on topics like object detection, motion tracking, path planning and game theory. Path planning has direct application in the navigation of a drone while object detection and tracking are crucial for obstacle avoidance. The part of game theory used can be coupled with path planning for decision making during flight.

INTRODUCTION

As a part of task round of selections into the software team of Aerial Robotics Kharagpur, I was given a set of tasks ranging from coding a Tic-Tac-Toe agent to detecting aircraft trajectories based on radar reading. Following are the problem statements, attempts, struggles, and outcomes of the first four tasks.

I. Task 2.1

PROBLEM STATEMENT

We are required to build an agent that plays the game of Tic-Tac-Toe using the minmax algorithm. The task provides a gym-tictactoe environment which consists of a modular code base to help us with the task.

INITIAL ATTEMPTS

I, first, spent 1 hour or so understanding the code base. I, then, read up minmax algorithm from the link shared. I was thinking of a method where I could call upon a function that builds up the game tree from every time when it's the AI's turn. But, I have noticed that it would be a waste of resources if I kept calling on the same region of the game tree. This led me to think of my final approach.

FINAL APPROACH

I wanted to generate the entire game tree to start with and then access the nodes as and when necessary. So I began making a class for a Node. I designed the node to have two important methods and one small method to make life easier. Here's the initialization of the Node class.

```
1  class Node():
2      def __init__(self, state, parent, ava_actions,
       level):
3          self.parent = parent
4          self.ava_actions = ava_actions
5          self.mark = state[1]
6          self.board = state[0]
7          self.value = 0
8          self.level = level # depth
```

The first important method is Node.fill(). This is basically a recursive method that goes depth first. At each step taken, the status of the game is checked. To do this, I made use of an aptly named method from code base called check_game_status. Upon checking if it turned out that the game ended in a win, loss or draw, then I would set the value of the node (which is now a leaf) taking into account how far down has the result come (Will be explained in further detail later). So, I would go around finding all the leafs and set their respective values. Let's look at the code so far.

```
1          # Filling the leafs
2          # For papa
3          if self.level == 0:
4              self.children = {}
5              print("Loading...\n10")
6              for move in self.ava_actions:
7                  nstate, _ava_actions = self.acting(
       move)
8                  print(10 - move)
9                  self.children[move] = Node(nstate,
       self, _ava_actions, self.level + 1)
10                 self.children[move].fill()
11
12         # For every other node
13         elif self.level >= 1 and self.level <= 9:
14             status = check_game_status(self.board)
15             if status == 2:
16                 self.value = 10 - self.level
17             elif status == 1:
18                 self.value = -10 + self.level
19             elif status == 0:
20                 self.value = 0
21             elif status == -1:
22                 self.children = {}
23                 for move in self.ava_actions:
24                     nstate, _ava_actions = self.
       acting(move)
25                     self.children[move] = Node(
       nstate, self, _ava_actions, self.level + 1)
26                     self.children[move].fill()
```

Lines 3 - 10: Papa is the root of the game tree. The depth of Papa is 0. For each available action, the list of remaining action is passed to the children of Papa whose fill function is called. I found the remaining actions by using the little function I created that is self.acting() (More on this later). Each child is stored in a dictionary with move taken to reach there being the key. Lines 6 and 8 are just for aesthetic purposes.

Lines 13 - 20: Here, there is a check whether the node under observation is a leaf. This is done from lines 15 - 20. It is checked whether the game is won (if status is 2), lost (if status is 1) or draw (if status is 0). Then, the value of that

* Archit Rungta, Yash Soni

leaf is set using the metric mentioned before. The metric is such that if the game is won even before it starts the score of 10. But, considering the depth of the tree, the value is 10 - (depth). The following diagram should explain it better. For every loss, the value is -10 + (depth). This is done to encourage closer victories and discourage nearer losses.

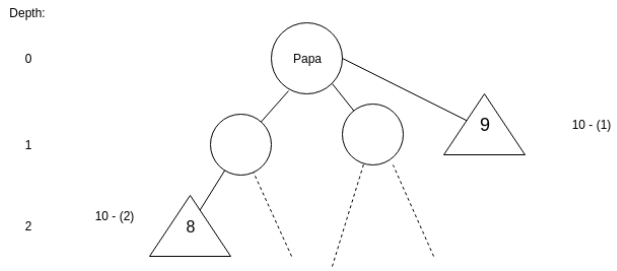

Fig. 1. Valuing Scheme

Lines 21 - 26: Here, I'm checking if the game is still incomplete (if status is -1). Then, I call upon its children like how I did with Papa and fill them up.

Now, moving to the next part of fill method that is filling every node.

```
        status = check_game_status(self.board)
        if status == -1:
            self.Max = self.children[self.
    ava_actions[0]].value
            self.maxAddress = self.ava_actions[0]
            self.Min = self.children[self.
    ava_actions[0]].value
            self.minAddress = self.ava_actions[0]
            for move in self.ava_actions:
                if self.Max < self.children[move].
    value:
                    self.Max = self.children[move].
    value
                    self.maxAddress = move

                if self.Min > self.children[move].
    value:
                    self.Min = self.children[move].
    value
                    self.minAddress = move

            if self.mark == 'O':
                self.value = self.Min
            if self.mark == 'X':
                self.value = self.Max
```

Lines 1 - 2: I, first, see whether if the node under consideration is not a leaf with status.

Lines 3 - 14: Then, I find the max and min values among the children.

Lines 16 - 20: Applying the main essence of minmax algorithm by setting the value of node as minimum if it is the human's turn ('O') and as maximum if it is the AI's turn ('X').

Considering the next important method of Node class is reach_child().

```
    def reach_child(self, move):
        return self.children[move].maxAddress, self
    .children[move].children[self.children[move].
    maxAddress]

```

This function is meant to return a move and an instance of a Node when given a move made by human while we are at a node where it is the AI's turn. The move returned is maxAddress. The value maxAddress of a node signifies the move that needs to be taken to reach the child with the maximum value. Also the node corresponding to this child is the instance of a Node which is returned.

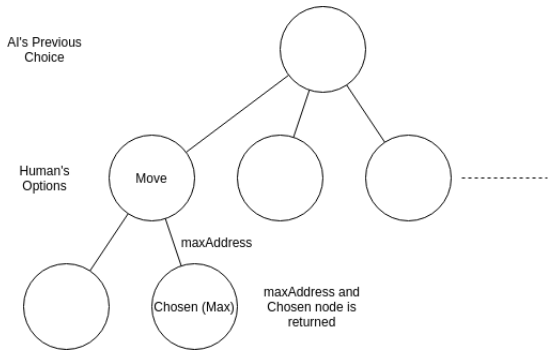

Fig. 2. reach_child() execution

The last method in Node class is acting(). This method takes in a move to be made and spits out the new state (nstate) and a new set of available actions (_ava_actions).

```
    def acting(self, move):
        _ava_actions = self.ava_actions[:]
        _ava_actions.remove(move)
        nboard = list(self.board)
        nboard[move - 1] = tocode(self.mark)
        if self.mark == 'O':
            nstate = (nboard, 'X')
        elif self.mark == 'X':
            nstate = (nboard, 'O')

        return nstate, _ava_actions
```

Lines 2 and 3: Copying the current set of actions and removing the move used to get to the new list available actions.

Lines 4 - 9: The move is marked on the board and new state with the new board and new next move is generated.

Now, let us see how I incorporated this game tree with the minmax_agent.py. I removed the in-built MinMaxAgent class and made the following changes to the play() method.

```
def play(show_number):

    env = TicTacToeEnv(show_number=show_number)
    agents = [HumanAgent(HUMAN_MARK)]
    episode = 0
    j = 0
    while True:

        state = env.reset()
        _, mark = state
        done = False
        env.render()
        i = 0
        if j == 0:
            Papa = Node(state, None, [1, 2, 3, 4,
    5, 6, 7, 8, 9], 0)
            Papa.fill()
            j += 1
        action = Papa.maxAddress
        current = Papa.children[Papa.maxAddress]
```

```
        print("X's Turn")
        while not done:
            pre_action = action
            pre_current = current
            ava_actions = env.available_actions()
            if i % 2 == 0 and i != 0:
                print("X's Turn")
                print("Previous Action: ",
    pre_action)
                action, current = pre_current.
    reach_child(pre_action)
                print("Playing: ", action)
            elif i % 2 == 1:
                print("O's Turn")
                action = HumanAgent.act(state,
    ava_actions)

            i += 1
            if action is None:
                sys.exit()

            state, reward, done, info = env.step(
    action - 1)

            print('')
            env.render()
            if done:
                env.show_result(True, mark, reward)
                break
            else:
                _, _ = state
        mark = next_mark(mark)

        episode += 1
```

Line 6 and 14 - 17: I am making sure that Papa is filled once every session of game.

Line 18 - 19: action is the move that is about to take place so I set the maxAddress of Papa as the first move. I also made the current node as the node corresponding to the first action.

Line 13 and 22 - 34: i is to alternate between human and AI. When it is the AI's turn, the reach_child method of the current node is called to give new action and set the new current. When it is the human's turn, input is taken from the act()method of HumanAgent.

RESULTS AND OBSERVATION

Given that the following is the layout of the game, it is found that the AI prefers to play in place 1 as the first move.

1	2	3
4	5	6
7	8	9

Provided that the human plays ideally, here are the results table based on the position where the human plays in the remaining spots.

2	Win
3	Win
4	Win
5	Draw
6	Win
7	Win
8	Win
9	Win

It is observed that AI never loses and draws only in one situation, proving that the minmax algorithm is very effective.

FUTURE WORK

During the implementation of the algorithm, I found some situations which the minmax algorithm may overlook.

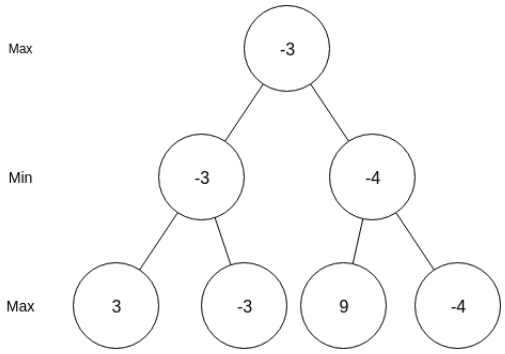

Fig. 3. Limitation of MinMax algorithm

Consider the above example, the topmost Max chooses '-3' believing it is the lesser evil with the assumption that Min would choose the worst possible option. But, the same case in human vs AI version, it may not be perfectly evident for the human that choosing '-4' is the better option for him/her. This is not the case with a simple game like Tic-Tac-Toe. So, I thought of capitalizing on this uncertainty by adding weights to different options and computing the probability of the human selecting '9' over '-4' and also between '3' and '-3' and then make the decision accordingly.

II. Task 2.2

PROBLEM STATEMENT

We are given an image with several circles and a line which represents the direction of motion of the circle attached to it. Ignoring oblique collisions and assuming perfectly elastic collisions between any two objects, we must simulate every collisions to come.

Here is an example image.

INITIAL ATTEMPTS

My first approach is the one I stuck with. Here it goes.

FINAL APPROACH

I have used pygame library to make the simulation. The files 'sim.py', 'sprite.py', 'settings.py' are related to the simulation and the file 'Read.py' is the one that detects the lines and circles of the image. I will only dwell into the details of 'Read.py'.

'Read.py' consists of a method LandC() which uses OpenCV tools to detect the circles and the lines.

```
def LandC(image):
    img = cv2.imread(image, 1)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    img1 = cv2.GaussianBlur(img, (7, 7), 0)

```

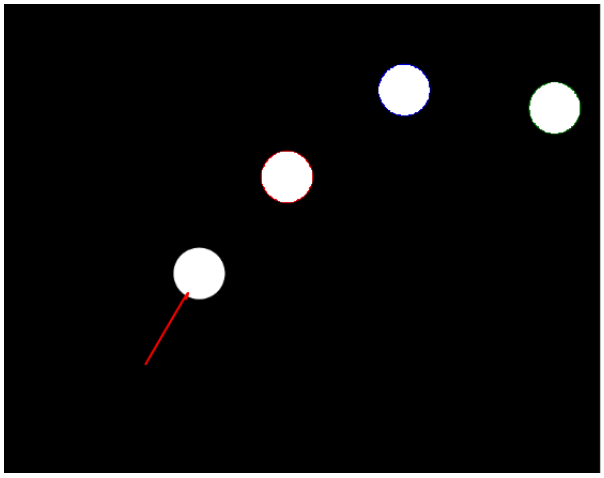

Fig. 4. Example Image

```
7     edges = cv2.Canny(img1, 75, 150)
8     lines = cv2.HoughLinesP(edges, 1, np.pi/180,
      40, maxLineGap=30)
9
10    X1, Y1, X2, Y2 = 0, 0, 0, 0
11    n = len(lines)
12    for line in lines:
13        x1, y1, x2, y2 = line[0]
14        X1 += x1/n
15        Y1 += y1/n
16        X2 += x2/n
17        Y2 += y2/n
18
19    X1 = int(X1)
20    Y1 = int(Y1)
21    X2 = int(X2)
22    Y2 = int(Y2)
23
24    HEIGHT, WIDTH = img.shape
25
26    all_circs = cv2.HoughCircles(img1, cv2.
      HOUGH_GRADIENT, 1, 120,
27                                 param1=100, param2
      =30, minRadius=0, maxRadius=0)
28    all_circs_rounded = np.uint16(np.around(
      all_circs))
29
30    circs = list(all_circs_rounded[0])
31    for i in range(len(circs)):
32        circs[i] = list(circs[i])
33
34    radians = math.atan2(Y1 - Y2, X1 - X2)
35
36    dx = math.cos(radians)
37    dy = math.sin(radians)
38
39    print(f"{radians} {dx} {dy}")
40
41    return dx, dy, circs, WIDTH, HEIGHT, X1, Y1, X2
      , Y2
```

Lines 2 - 5: Reading the image and converting to grayscale and blurring it. See Fig. 5 and Fig. 6.

Lines 7: Using Canny edge detector to find the edges. See Fig. 7.

Lines 10 - 22: Using HoughLinesP() method to find the lines in the picture and averaging it out. See Fig. 8.

Lines 26 - 28: Detecting the circles and round the co-ordinates. Detection is done using the HoughCircles() method. Sorry, Image not available.

Lines 34 - 38 and 41: Determining the inclination of the lines with the axes and returning all the necessary quantities.

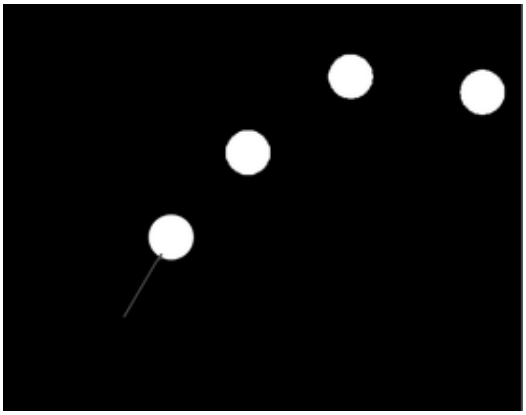

Fig. 5. Grayscale Image

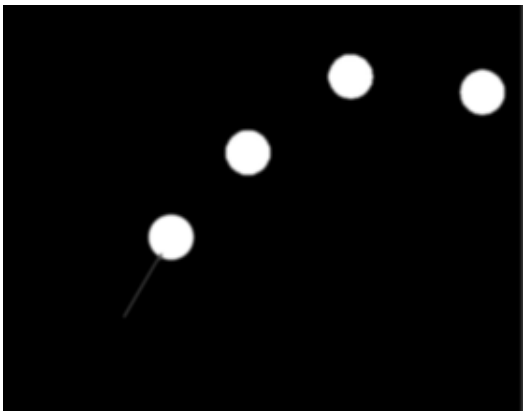

Fig. 6. Blurred Image

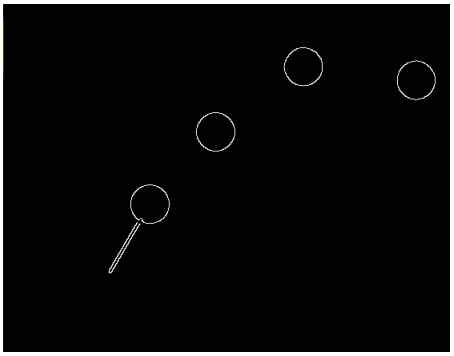

Fig. 7. Applying Canny edge Detector

RESULTS AND OBSERVATION

Result turned out as expected. With the help of pygame library I was able to make a smooth simulation. The simulation will be shown during the demonstration. There have been some hiccups with some of the test cases which I believe can be resolved by changing the threshold values.

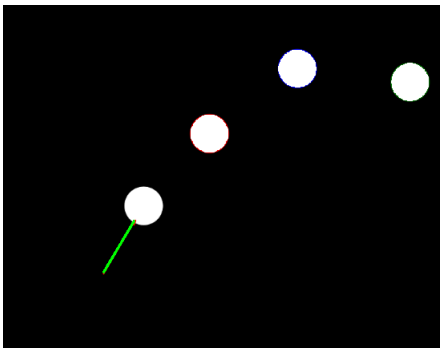

Fig. 8. Line detected

FUTURE WORK

One improvement that can be made is allowing for oblique collisions and inelastic collisions. The ideas used in the task can be applied detecting obstacles and predict their motions to avoid it in time.

III. Task 2.3

PROBLEM STATEMENT

The task requires us to create a face tracking system. Using this tracking system, we should be able to navigate a sprite through obstacles in a game.

INITIAL ATTEMPTS

Again, my initial attempt was my final one as well. Here it is.

FINAL APPROACH

Like before I have made use the pygame library to make my game. The files 'main.py', 'settings.py' and 'sprites.py' are responsible for running the game while 'face.py' is for tracking the face movement. 'Pipe' helps transferring commands from 'face.py' to 'sprites.py'. Let's us look into the details of 'face.py'.

```
while True:
    # Capture frame-by-frame
    ret, frame = video_capture.read()

    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    faces = faceCascade.detectMultiScale(
        gray,
        scaleFactor=1.1,
        minNeighbors=5,
        minSize=(30, 30),
        flags=cv2.FONT_HERSHEY_SIMPLEX
    )

    eyes = eyeCascade.detectMultiScale(
        gray,
        scaleFactor=1.1,
        minNeighbors=5,
        minSize=(30, 30),
        flags=cv2.FONT_HERSHEY_SIMPLEX
    )
```

Lines 3 and 5: Captures the video from the webcam frame by frame and converts each frame to gray-scale.

Lines 7 - 21: Makes use cascade classifiers ("haarcascade_frontalface_alt.xml", "haarcascade_eye.xml") to detect faces and eyes in each frame.

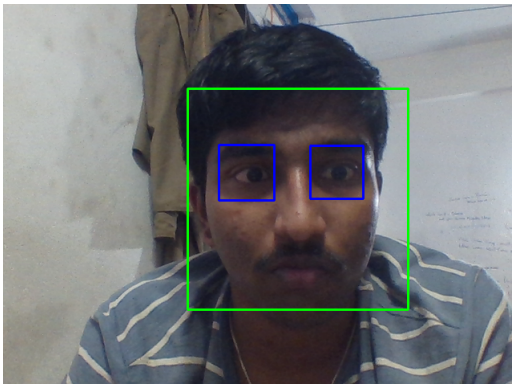

Fig. 9. Face and Eyes Detected

```
    if i < 100:
        if len(faces) > 0:
            fuavg += (faces[0][1] + faces[0][3]) /
    100
            fdavg += faces[0][1] / 100
            i += 1

    if j < 100:
        if len(eyes) > 0:
            euavg += (eyes[0][1] + eyes[0][3]) /
    100
            edavg += (eyes[0][1]) / 100
            j += 1

    if i == 100 and k == 0 and j == 100:
        favg = round((fuavg + fdavg) / 2)
        eavg = round((euavg + edavg) / 2)
        print(f"Face upper avg {fuavg}")
        print(f"Face lower avg {fdavg}")
        print(f"Eye upper avg {euavg}")
        print(f"Eye lower avg {edavg}")
        print(f"Face avg {favg}")
        print(f"Eye avg {eavg}")
        k += 1
```

This segment, with the help of counters and flags, scans the face and eyes 100 times and computes the average of the location of centers of the face and eyes to get a mean position about which we can sense deviation to track if the face moves up or down.

```
    if k == 1:
        if len(eyes) > 0:
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break
            cure = (eyes[0][1] + eyes[0][3] + eyes
    [0][1]) / 2
            action = isHigher(cure, eavg)
            print(action[1])
            with open("Pipe", 'w') as f:
                f.write(action[1])

        elif len(faces) > 0:
            curf = (faces[0][1] + faces[0][3] +
    faces[0][1]) / 2
            action = isHigher(curf, favg)
            print(action[1])
            with open("Pipe", 'w') as f:
                f.write(action[1])
```

In this segment, I am measuring current eye and current face position as cure and curf. I noticed that the face detection varies a lot. So, I first check which direction the eyes have displaced. If in any case the eye detection fails, then face displacement. The displacement is sent to the 'Pipe' for 'sprites.py' to use. The displacement is checked by the following method.

```
def isHigher(a, b):

    if a > b + 10:
        return a - b - 10, "DOWN"
    elif a < b - 10:
        return b - 10 - a, "UP"
    else:
        return b, "SAME"
```

Here, I am comparing heights of 'a' and 'b'. The code speaks for itself.

RESULTS AND OBSERVATION

I am able to detect motion of the face and eye and control the game. I have noticed a slight delay between the movement of the face and the sprite. Video of working will be provided at the demonstration.

FUTURE WORK

An enhancement to the present state of the game could be to make the obstacles in many more orientations. I could also try to implement some proper object tracking algorithms like template matching.

IV. Task 3.1

PROBLEM STATEMENT

The task requires us to build a strategy to control a unit in a combat game. We are provided with a game environment with a large code base. The task has two parts - guiding the unit to a weapon and shooting the enemy.

INITIAL ATTEMPTS

I spent a significant amount of time understanding the code flow and learnt about various path planning algorithms. My first attempt of executing Rapidly-exploring Random Tree (RRT) (Not 100 per cent) was how far I could get. So here it is.

FINAL APPROACH

Based on the time constraints, the movement that is allowed in one tick is as below. As there is no fractional vertical movement, they are three distinct line segments.

But for simplification of things I considered that possible states in horizontal direction are integral values as shown below.

I only mentioned vertical movement and horizontal movement in the diagram because magnitudes can vary with game constants and the vertical movement, especially, changes based on whether the unit is on a platform or on a jump pad. Keeping in mind these conditions, let's see my attempt at implementing RRT.

First, I noticed there would a necessity for a method that can say if two points have a line of sight. Hence, I created the los() method.

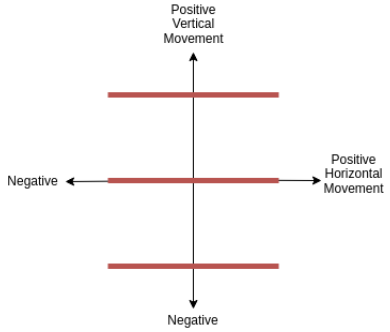

Fig. 10. Ideal Conditions

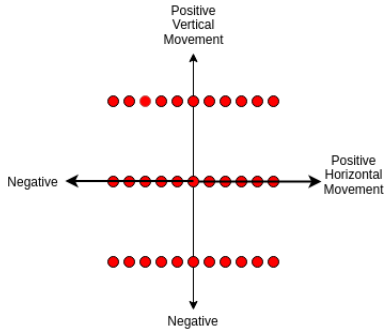

Fig. 11. Assumed Conditions

```
    def los(self, pos1, pos2, debug):
        if not (pos1.x < pos2.x or pos1.y < pos2.y)
    :
            pos1, pos2 = pos1, pos2
        width = round(pos2.x - pos1.x)
        height = round(pos2.y - pos1.y)
        max_ = max(width, height)
        if max_ != 0:
            _height = height / max_
            _width = width / max_
            for i in range(max_):

                if self.game.level.tiles[round(pos1
    .x + i * _width)][round(pos1.y + i * _height)]
    == 1:
                    return False
            return True
        return True
```

Lines 2 and 3: I am making sure neither width nor height turn out negative.

Lines 4 - 15: I am moving from one point to other after calculating how much width and height I have to move to form an approximate line. I am ensuring that no wall appears at each step and when I reach the destination successfully I return 'True', else 'False'.

I, then, created a class for a node. Let's discuss it method by method.

```
class Node():
    def __init__(self, unit, parent, game):
        self.unit = unit
        self.pos = unit.position
        self.game = game
        self.jump_state = unit.jump_state
        self.parent = parent
        self.prop = self.game.properties
```

```
        self.stateX = [0, 1, -1, 2, -2, 3, -3, 4, -4, 5, -5, 6, -6, 7, -7, 8, -8, 9, -9, 10, -10]
        self.stateY = [0, 1, -1]
        self.kids = []
        self.go = 0
```

In the initialization I brought in all necessary instances of different objects. 'self.stateX' is actually hard-coded. It is the discrete version of all possible horizontal velocities.

```
    def get_cords(self, xstate, ystate):
        time = 1/(self.prop.ticks_per_second * self.prop.updates_per_tick)
        if ystate == 1:
            height = self.prop.unit_jump_speed / time
            if self.game.level.tiles[floor(self.pos.x)][floor(self.pos.y)] == 4:
                height = self.prop.jump_pad_jump_speed / time
        if ystate == -1:
            height = self.prop.unit_fall_speed / time
        if ystate == 0:
            height = 0

        width = self.prop.unit_max_horizontal_speed / (time)

        return model.Vec2Double(self.pos.x + width * xstate, self.pos.y + height * ystate)
```

The above method was created with the intention that I would need to find out the co-ordinates of possible states based on the position of the node. It accounts for the difference between the vertical velocities based on the type of block the unit is standing upon. It also accounts for the time spent per each update and then moving accordingly to get coordinates.

```
    def pick(self, xsamp, strat):
        self.debug = strat.debug
        min_ = inf
        for x in self.stateX:
            for y in self.stateY:
                if min_ > comp(xsamp, self.get_cords(x, y)):
                    min_ = comp(xsamp, self.get_cords(x, y))
                    xstate = x
                    ystate = y

        if strat.los(self.get_cords(xstate, ystate), self.pos, self.debug) == True:
            n = Node(self.unit, self, self.game)
            n.xstate = xstate
            n.ystate = ystate
            self.kids.append(n)
            strat.tree.append(n)
```

When the sample point is chosen from the free space, every possible state's co-ordinates is checked with node original position to check for line of sight and minimum distance from the sample point. All of this is done only when this particular node is selected as the nearest node in the tree to the sample point.

```
    def set(self, debug):
        self.go = 1
        if self.parent != None:
            self.parent.set()
        else:
            return self.execute(debug)
```

This recursive method was implemented to let each parent know which child should be used to reach the destination. So, when it is known that the destination is found, every child will inform their parents where to head when it is their turn to move. After notifying, it initializes the movement.

```
    def execute(self, debug):
        for kid in self.kids:
            debug.draw(model.CustomData.Log(f"{kid.xstate} {kid.ystate}"))
            if kid.go == 1:
                debug.draw(model.CustomData.Log(f"Entered"))
                hvel = kid.xstate
                if kid.ystate == 0:
                    ujump = False
                    djump = False
                if kid.ystate == 1:
                    ujump = True
                    djump = False
                if kid.ystate == -1:
                    ujump = False
                    djump = True
                debug.draw(model.CustomData.Log(f"{hvel} {ujump} {djump}"))
                return hvel, ujump, djump
```

Coming to the last method of Node class which is execute(). It checks which kid has their 'go' value as 1, and then use that child's state values and return set of actions that need to be taken.

```
    def RRT(self, start_pos, end_pos):
        self.tree = [Node(self.unit, None, self.game)]
        goal = [model.Vec2Double(floor(end_pos.x), floor(end_pos.y)), model.Vec2Double(floor(end_pos.x + 1), floor(
            end_pos.y)), model.Vec2Double(floor(end_pos.x), floor(end_pos.y + 1)), model.Vec2Double(floor(end_pos.x + 1), floor(end_pos.y + 1))]

        while len(self.tree) != 1000:
            xsamp = model.Vec2Double(randrange(1, 40), randrange(1, 30))

            dist = [comp(xsamp, n.pos) for n in self.tree]

            near = dist.index(min(dist))
            nearNode = self.tree[near]
            nearNode.pick(xsamp, self)

            leaf = model.Vec2Double(floor(self.tree[-1].pos.x), floor(self.tree[-1].pos.y))
            if leaf in goal:
                return self.tree[-1].set(self.debug)
```

Finally, RRT itself.

Lines 2 - 4: With the starting and ending points at hand, we begin the tree and set the goal as a 2 X 2 region.

Lines 5 - 17: A loop that runs until there are 1000 nodes in the tree where the following happens on each iteration.

- A random point is picked in the free space.
- The nearest node to the random point is found.
- The likely state in that node is chosen by calling its pick function
- The latest node is checked if it lies in the goal region which if true, the entire setting train is activated and

action is spit out.

Following this is just entering this found action into the default code given to see the outcome.

RESULTS AND OBSERVATION

Unfortunately, I could not see any outcome. The TPS (Ticks Per Second) dropped very low and no movement of the unit was observed. I feel that each step is too small and sufficient nodes aren't being generated. Increasing the number of allowed nodes in the tree only makes the game even more slower.

FUTURE WORK

I think there can be way of visualising the state space can be improved so the path can be found in much fewer steps. Or I could break the motion to many smaller parts and try. One thing I did think of but didn't implement is considering the jump time limiting the time vertical motion can happen.

CONCLUSION

This whole month has been great that I've got so many new things to learn. I have never seen myself this dedicated before. Sorry for (m)any errors in the documentation. I didn't have sufficient time to proof-read it. Tasks 2.2 and 2.3 were easy to solve. But Tasks 2.1 and 3.1 twisted my brain a lot.

Also apologies for no bibliography and related work section and not commenting the code.