**A-star Breakdown**

by Ken

if \_\_name\_\_ == '\_\_main\_\_':

scenario = input("Enter 1/2/3/4 for scenario1/2/3/task2 respectivelly:")

main(scenario)

# The key line of the code

Line 419-421 , is the most important line in the hole code,

which act as a detector to check whether the current running file name is the is its own name(for example, importing files) .

If the current file name is its own name, then execute line 420-421. In line 420, **“input()”** function is used to capture a **string** for indicating the scenario. In line 421, is the most important part that it drives the code to run the **“main()”**function with carrying a variable **“scenario”**(a string)

# What is “main()” function

In line 284, it is the definition of **“main()”**. It require user to input a variable during the usage, in later part the input was use as a indicator to direct and run the corresponding code for different task and scenario.

Line287-294, set variable.

sx: starting point x-coordinate

sy: starting point y-coordinate

gx: goal point x-coordinate

gy: goal point y-coordinate

Tbest: time-best

line 298-321: set the coordinate location of the obstacle

line 330-349: set the coordination of cost intensive area

note:in line 344: set the limitation of existence for jet-steam area

def main(scenario):

print(\_\_file\_\_ + " start the A star algorithm demo !!") # print simple notes

# start and goal position

sx = 50.0 # [m]

sy = 32.0 # [m]

gx = -5.0 # [m]

gy = 0.0 # [m]

grid\_size = 1 # [m]

robot\_radius = 1.0 # [m]

Tbest = 0

# set obstacle positions for group 8

ox, oy = [], []

for i in range(-10, 60): # draw the buttom border

ox.append(i)

oy.append(-10.0)

for i in range(-10, 60): # draw the right border

ox.append(60.0)

oy.append(i)

for i in range(-10, 60): # draw the top border

ox.append(i)

oy.append(60.0)

for i in range(-10, 60): # draw the left border

ox.append(-10.0)

oy.append(i)

for i in range(20, 60): # draw the free border

ox.append(40.0)

oy.append(i)

for i in range(0, 10):

ox.append(i)

oy.append(-2 \* i + 10)

for i in range(40, 60):

ox.append(0)

oy.append(i)

# for i in range(40, 45): # draw the button border

# ox.append(i)

# oy.append(30.0)

# set cost intesive area 1 red in colour i=horizontal j=vertical

tc\_x, tc\_y = [], []

for i in range(10, 30):

for j in range(10, 40):

tc\_x.append(i)

tc\_y.append(j)

# set cost intesive area 2 yellow in colour

fc\_x, fc\_y = [], []

for i in range(35, 50):

for j in range(0, 10):

fc\_x.append(i)

fc\_y.append(j)

jc\_x, jc\_y = [], []

if scenario == "4" :

# set cost intesive area 3(jet-steam) cyan in colour(i:horizontal , j:vertical)magenta

for i in range(-10, 60):

for j in range(9, 14):

jc\_x.append(i)

jc\_y.append(j)

if show\_animation: # pragma: no cover

plt.plot(ox, oy, ".k") # plot the obstacle

plt.plot(sx, sy, "\*b") # plot the start position

plt.plot(gx, gy, "\*g") # plot the end position

plt.plot(fc\_x, fc\_y, "oy") # plot the cost intensive area 2(fule) yellow

plt.plot(tc\_x, tc\_y, "or") # plot the cost intensive area 1(time) red

plt.plot(jc\_x, jc\_y, "\_m") # plot the cost intensive area 3(jet-stream) magenta

plt.grid(True) # plot the grid to the plot panel

plt.axis("equal") # set the same resolution for x and y axis

a\_star = AStarPlanner(ox, oy, grid\_size, robot\_radius, fc\_x, fc\_y, tc\_x, tc\_y, jc\_x, jc\_y)

rx, ry, Tbest = a\_star.planning(sx, sy, gx, gy)

#print(Tbest)

#calc for the input scenario

if scenario == '1' :

Cost\_A321 = (0.76\*54\*Tbest+15\*Tbest+1800)\*15

Cost\_A330 = (0.76\*84\*Tbest+21\*Tbest+2000)\*10

Cost\_A350 = (0.76\*90\*Tbest+27\*Tbest+2500)\*9

print("The Trip Cost for A321neo is ${}".format(round(Cost\_A321)))

print("The Trip Cost for A330-900neo is ${}".format(round(Cost\_A330)))

print("The Trip Cost for A350-900 is ${}".format(round(Cost\_A350)))

print("A330-900neo has the lowest Trip Cost among them.")

elif scenario == '2' :

Cost\_A321 = (0.88\*54\*Tbest+20\*Tbest+1800)\*7

Cost\_A330 = (0.88\*84\*Tbest+27\*Tbest+2000)\*5

Cost\_A350 = (0.88\*90\*Tbest+34\*Tbest+2500)\*4

print("The Trip Cost for A321neo is ${}".format(round(Cost\_A321)))

print("The Trip Cost for A330-900neo is ${}".format(round(Cost\_A330)))

print("The Trip Cost for A350-900 is ${}".format(round(Cost\_A350)))

print("A350-900 has the lowest Trip Cost among them.")

elif scenario == '3' :

Cost\_A321 = (0.95\*54\*Tbest+10\*Tbest+1800)\*13

Cost\_A330 = (0.95\*84\*Tbest+15\*Tbest+2000)\*9

Cost\_A350 = (0.95\*90\*Tbest+20\*Tbest+2500)\*7

print("The Trip Cost for A321neo is ${}".format(round(Cost\_A321)))

print("The Trip Cost for A330-900neo is ${}".format(round(Cost\_A330)))

print("The Trip Cost for A350-900 is ${}".format(round(Cost\_A350)))

print("A3500-900 has the lowest Trip Cost among them.")

elif scenario == '4' :

Cost\_A321 = (0.76\*54\*Tbest+15\*Tbest+1800)\*15

Cost\_A330 = (0.76\*84\*Tbest+21\*Tbest+2000)\*10

Cost\_A350 = (0.76\*90\*Tbest+27\*Tbest+2500)\*9

print("The Trip Cost for A321neo is ${}".format(round(Cost\_A321)))

print("The Trip Cost for A330-900neo is ${}".format(round(Cost\_A330)))

print("The Trip Cost for A350-900 is ${}".format(round(Cost\_A350)))

print("A330-900neo has the lowest Trip Cost among them.")

else :

print("There are no scenario {}, plz enter a correct one. Thank You".format(scenario))

if show\_animation: # pragma: no cover

plt.plot(rx, ry, "-r") # show the route

plt.pause(0.001) # pause 0.001 seconds

plt.show() # show the plot

jc\_x, jc\_y = [], []

if scenario == "4" :

# set cost intesive area 3(jet-steam) cyan in colour(i:horizontal , j:vertical)magenta

for i in range(-10, 60):

for j in range(9, 14):

jc\_x.append(i)

jc\_y.append(j)

# con’d main( ) function

line352-362: plot the grath(obstacle, start-end point, cost intensive area)

if show\_animation: # pragma: no cover

plt.plot(ox, oy, ".k") # plot the obstacle

plt.plot(sx, sy, "\*b") # plot the start position

plt.plot(gx, gy, "\*g") # plot the end position

plt.plot(fc\_x, fc\_y, "oy") # plot the cost intensive area 2(fule) yellow

plt.plot(tc\_x, tc\_y, "or") # plot the cost intensive area 1(time) red

plt.plot(jc\_x, jc\_y, "\_m") # plot the cost intensive area 3(jet-stream) magenta

plt.grid(True) # plot the grid to the plot panel

plt.axis("equal") # set the same resolution for x and y axis

line364-365: path finding (we cover it in lower pages)

a\_star = AStarPlanner(ox, oy, grid\_size, robot\_radius, fc\_x, fc\_y, tc\_x, tc\_y, jc\_x, jc\_y)

rx, ry, Tbest = a\_star.planning(sx, sy, gx, gy)

#print(Tbest)

line369-410: print corresponding information for each task and scenario

#calc for the input scenario

if scenario == '1' :

Cost\_A321 = (0.76\*54\*Tbest+15\*Tbest+1800)\*15

Cost\_A330 = (0.76\*84\*Tbest+21\*Tbest+2000)\*10

Cost\_A350 = (0.76\*90\*Tbest+27\*Tbest+2500)\*9

print("The Trip Cost for A321neo is ${}".format(round(Cost\_A321)))

print("The Trip Cost for A330-900neo is ${}".format(round(Cost\_A330)))

print("The Trip Cost for A350-900 is ${}".format(round(Cost\_A350)))

print("A330-900neo has the lowest Trip Cost among them.")

elif scenario == '2' :

Cost\_A321 = (0.88\*54\*Tbest+20\*Tbest+1800)\*7

Cost\_A330 = (0.88\*84\*Tbest+27\*Tbest+2000)\*5

Cost\_A350 = (0.88\*90\*Tbest+34\*Tbest+2500)\*4

print("The Trip Cost for A321neo is ${}".format(round(Cost\_A321)))

print("The Trip Cost for A330-900neo is ${}".format(round(Cost\_A330)))

print("The Trip Cost for A350-900 is ${}".format(round(Cost\_A350)))

print("A350-900 has the lowest Trip Cost among them.")

elif scenario == '3' :

Cost\_A321 = (0.95\*54\*Tbest+10\*Tbest+1800)\*13

Cost\_A330 = (0.95\*84\*Tbest+15\*Tbest+2000)\*9

Cost\_A350 = (0.95\*90\*Tbest+20\*Tbest+2500)\*7

print("The Trip Cost for A321neo is ${}".format(round(Cost\_A321)))

print("The Trip Cost for A330-900neo is ${}".format(round(Cost\_A330)))

print("The Trip Cost for A350-900 is ${}".format(round(Cost\_A350)))

print("A3500-900 has the lowest Trip Cost among them.")

elif scenario == '4' :

Cost\_A321 = (0.76\*54\*Tbest+15\*Tbest+1800)\*15

Cost\_A330 = (0.76\*84\*Tbest+21\*Tbest+2000)\*10

Cost\_A350 = (0.76\*90\*Tbest+27\*Tbest+2500)\*9

print("The Trip Cost for A321neo is ${}".format(round(Cost\_A321)))

print("The Trip Cost for A330-900neo is ${}".format(round(Cost\_A330)))

print("The Trip Cost for A350-900 is ${}".format(round(Cost\_A350)))

print("A330-900neo has the lowest Trip Cost among them.")

else :

print("There are no scenario {}, plz enter a correct one. Thank You".format(scenario))

line412-415: to plot the graph that look like a animation

if show\_animation: # pragma: no cover

plt.plot(rx, ry, "-r") # show the route

plt.pause(0.001) # pause 0.001 seconds

plt.show() # show the plot

# Path Finding - AStarPlanner

“AStarPlanner” is a set of function, which covered from line23 to line281. it contain 3 parts (1.variable defining)(2.”Node”defining )(3.”planning( )”function)

## variable defining

line 35-55,

variable for map operation

self.resolution = resolution # get

resolution of the grid

self.rr = rr # robot radis

self.min\_x, self.min\_y = 0, 0

self.max\_x, self.max\_y = 0, 0

self.obstacle\_map = None

self.x\_width, self.y\_width = 0, 0

self.motion = self.get\_motion\_model() # motion model for grid search expansion

self.calc\_obstacle\_map(ox, oy)

variable for cost intensive area

self.fc\_x = fc\_x

self.fc\_y = fc\_y

self.tc\_x = tc\_x

self.tc\_y = tc\_y

self.jc\_x = jc\_x #add variables for Task2 which set to be scenario 4

self.jc\_y = jc\_y

self.Delta\_C1 = 0.2 # cost intensive area 1(time) modifier |yellow in colour

self.Delta\_C2 = 0.4 # cost intensive area 2(fule) modifier |red in colour

self.Delta\_C3 = -0.05 # cost intensive area 3(jet steam) |Magenta in colour

self.costPerGrid = 1 #Cost

line 178-281: A bunch of function are defined

“calc\_final\_plath” give a set of coordinates

“calc\_heuristic” give the heuristic cost value

“calc\_heuristic\_maladis” give a heuristic distance value

“calc\_grid\_position” convert the input to a grid position

“calc\_xy\_indes” give a value of a point on the plot

“calc\_grid\_index” give a value represent the grid-index

“verify\_nodes” check whether the node is possible

“calc\_obstacle\_map” generate the obstacle location on the plot

“get\_motion\_mode” set the possible movement for finding path

## “Node” defining

it bundle information in a format (we use this format down below)

class Node: # definition of a sinle node

def \_\_init\_\_(self, x, y, cost, parent\_index):

self.x = x # index of grid

self.y = y # index of grid

self.cost = cost

self.parent\_index = parent\_index

## “planning( )” function

It requires 4 input variable whit a specific sequence .

It covers form line84-176and represent the working principle of path finding.

line84-85: convert information of start-end point to Node format

start\_node =

self.Node(self.calc\_xy\_index(sx, self.min\_x), # calculate the index based on given position

self.calc\_xy\_index(sy, self.min\_y), 0.0, -1) # set cost zero, set parent index -1

goal\_node = self.Node(self.calc\_xy\_index(gx, self.min\_x), # calculate the index based on given position

self.calc\_xy\_index(gy, self.min\_y), 0.0, -1)

line89-90: create and fill in dictionary for “open\_set”,”closed\_set”

open\_set, closed\_set = dict(), dict() #

open\_set: node not been tranversed yet. closed\_set: node have been tranversed already

open\_set[self.calc\_grid\_index(start\_node)] = start\_node # node index is the grid index

noted that it is a For Loop “for true:”

while 1:

and output in a sequent of “rx”,”ry”,”goal\_node.cost”, which is important for us to capture corresponding info

return rx, ry, goal\_node.cost

line93: to test whether the situation is reachable

if len(open\_set) == 0:

print("Open set is empty..")

break

line97-102: set the current location and update it

c\_id = min(

open\_set,

key=lambda o: open\_set[o].cost + self.calc\_heuristic(self, goal\_node,open\_set[o])) # g(n) and h(n): calculate the distance between the goal node and openset

current = open\_set[c\_id]

line105-113: plot possible path (light-blue “X”) in animation format

# show graph

if show\_animation: # pragma: no cover

plt.plot(self.calc\_grid\_position(current.x, self.min\_x),self.calc\_grid\_position(current.y, self.min\_y), "xc")

# for stopping simulation with the esc key.

plt.gcf().canvas.mpl\_connect('key\_release\_event',

lambda event: [exit(

0) if event.key == 'escape' else None])

if len(closed\_set.keys()) % 10 == 0:

plt.pause(0.001)

line116-120: test whether the current situation touched the end point

# reaching goal

if current.x == goal\_node.x and current.y == goal\_node.y:

print("Total Trip time required -> ",current.cost )

goal\_node.parent\_index = current.parent\_index

goal\_node.cost = current.cost

break

line122-126: transfer the current location to the history list

# Remove the item from the open set

del open\_set[c\_id]

# Add it to the closed set

closed\_set[c\_id] = current

line131-170: a For Loop

line131: get possible movement data

for i, \_ in enumerate(self.motion): # tranverse the

motion matrix

line132-134: give a potential movement node information

node = self.Node(current.x +

self.motion[i][0],

current.y + self.motion[i][1],

current.cost + self.motion[i][2] \* self.costPerGrid, c\_id)

line136-152: adjust the time cost if the potential movement is touching cost intensive area

## add more cost in cost intensive

area 1

if self.calc\_grid\_position(node.x, self.min\_x) in self.tc\_x:

if self.calc\_grid\_position(node.y, self.min\_y) in self.tc\_y:

# print("cost intensive area!!")

node.cost = node.cost + self.Delta\_C1 \* self.motion[i][2]

# add more cost in cost intensive area 2

if self.calc\_grid\_position(node.x, self.min\_x) in self.fc\_x:

if self.calc\_grid\_position(node.y, self.min\_y) in self.fc\_y:

# print("cost intensive area!!")

node.cost = node.cost + self.Delta\_C2 \* self.motion[i][2]

# add more cost in cost intensive area 3

if self.calc\_grid\_position(node.x, self.min\_x) in self.jc\_x:

if self.calc\_grid\_position(node.y, self.min\_y) in self.jc\_y:

# print("cost intensive area!!")

node.cost = node.cost + self.Delta\_C3 \* self.motion[i][2]

line156: get potential movement node coordinates in the grid

n\_id = self.calc\_grid\_index(node)

line159-170: check the movement whether is possible or not and ignore those are not possible

# If the node is not safe, do

nothing

if not self.verify\_node(node):

continue

if n\_id in closed\_set:

continue

if n\_id not in open\_set:

open\_set[n\_id] = node # discovered a new node

else:

if open\_set[n\_id].cost > node.cost:

# This path is the best until now. record it

open\_set[n\_id] = node

The line131-170 For Loop end here!!!

line172: get the list of potential movement coordinates

rx, ry = self.calc\_final\_path(goal\_node,

closed\_set)

line176: output some information above in a specific sequence

return rx, ry, goal\_node.cost

**End of the For Loop from line92!!!**

noted for each run of the loop just calculate what next possible steps **could be**.

Hope this can help you a lot ^.^

Let’s review the main( )

At start: some variable is sat

Then: set obstacle location and cost intensive area

Next: plot those obstacle and area

After that: execute AStarPlanner

Moreover: bast on the scenario print specific info

Finally: plot the path