```
 1 # This is a sample Python script.
 2
 3 # Press Shift+F10 to execute it or replace it with your code.
 4 # Press Double Shift to search everywhere for classes, files, tool windows, actions
   , and settings.
 5 import cv2
 6 import numpy as np
 7 import copy
 8
 9 def move_up(cost_to_come,parent_node): #Applies up move, if there is obstacle
   returns original node
10     x = parent_node[0]
11     y = parent_node[1]
12     move = y+1
13     up = ((x,move), 1.0)
14     return up
15
16 def move_up_right(cost_to_come,parent_node): #Applies up_right move, if there is
   obstacle returns original node
17     x = parent_node[0]
18     y = parent_node[1]
19     movex = x + 1
20     movey = y + 1
21     up_right = ((movex, movey),1.4)
22     return up_right
23
24 def move_right(cost_to_come,parent_node): #moves blank right after checking for
```

```
24 obstacle, if there is obstacle returns original node
25     x = parent_node[0]
26     y = parent_node[1]
27     move = x + 1
28     right = ((move,y), 1.0)
29     return right
30
31 def move_down_right(cost_to_come,parent_node): #moves blank right after checking for
    obstacle, if there is obstacle returns original node
32     x = parent_node[0]
33     y = parent_node[1]
34     movex = x + 1
35     movey= y-1
36     down_right = ((movex, movey),1.4)
37     return down_right
38
39 def move_down(cost_to_come,parent_node): #moves blank right after checking for
   obstacle, if there is obstacle returns original node
40     x = parent_node[0]
41     y = parent_node[1]
42     movey= y-1
43     down = ((x, movey), 1.0)
44     return down
45
46 def move_down_left(cost_to_come,parent_node): #moves blank right after checking for
   obstacle, if there is obstacle returns original node
47     x = parent_node[0]
```

```
48      y = parent_node[1]
49      movey= y-1
50      movex= x-1
51      down_left = ((movex, movey),1.4)
52      return down_left
53
54  def move_left(cost_to_come,parent_node): #moves blank left after checking for
    obstacle, if there is obstacle returns original node
55      x = parent_node[0]
56      y = parent_node[1]
57      movex = x - 1
58      left = ((movex, y), 1.0)
59      return left
60
61  def move_up_left(cost_to_come,parent_node): #moves blank left after checking for
    obstacle, if there is obstacle returns original node
62      x = parent_node[0]
63      y = parent_node[1]
64      movex = x - 1
65      movey = y+1
66      up_left = ((movex, movey),1.4)
67      return up_left
68
69  def generate_path(reverse_path):
70      next_node = []
71      while next_node != "N/A":
72          search_for = reverse_path[-1]
```

```
73          reverse_path.append(cost_to_come[search_for]['parent node'])
74          next_node = cost_to_come[search_for]['parent node']
75
76      print("This is the reverse path from goal to start", reverse_path)
77
78      # This loop creates the forward path to goal
79      t = 0
80      forward_path = []
81      for t in range(len(reverse_path)):
82          forward_path.append(reverse_path.pop(-1))
83
84      # # this eliminates the start node from forward_path
85      forward_path.pop(0)
86      return forward_path
87
88  def check_for_goal(parent_node, goal_node):
89      SolutionFound=False
90      if parent_node == goal_node:
91          SolutionFound=True
92      return SolutionFound
93
94
95  cost_to_come=[]
96  visual_map=np.zeros((250, 600, 3), np.uint8)
97  visual_map[0:250,0:600,:] = [0,0,255]
98
99  x=0
```

```
y=0
node = 0
cost_to_come={}
for x in range(600):
    for y in range(250):
        node=(x,y)
        cost_to_come[node]={'x':0, 'y':0, 'parent node':"N/A", 'cost to come':0}
#cost_to_come = [node_id, x, y, parent node, cost to come]
        if (x >= 95) and (x <= 155) and (y >= 0) and (y <= 105):  #Obstacle A check
            cost_to_come[node]={'x': x,'y': y,'parent node': "N/A", 'cost to come'
:-1.0}
        else:
            if (x >= 95) and (x <= 155) and (y >= 145) and (y <= 250): #Obstacle B
check
                cost_to_come[node]={'x': x,'y': y,'parent node': "N/A", 'cost to
come':-1.0}
            else:
                #Obstacle C1 check
                if (x >= 300 - 37.5*3**0.5-5) and (x <= 300) and (y >= -(1/3)**(0.5
)*(x-300)+(125-75-50**(0.5))) and (y <= (1/3)**(0.5)*(x-300)+(125+75+50**(0.5))):
                    cost_to_come[node]={'x': x,'y': y,'parent node': "N/A", 'cost
to come':-1.0}
                else:
                    #Obstacle C2 check
                    if (x <= 300 + 37.5*3**0.5+5) and (x >= 300) and (y >= (1/3)**(
0.5)*(x-300)+(125-75-50**(0.5))) and (y <= -(1/3)**(0.5)*(x-300)+(125+75+50**(0.5
))):
```

```
120                         cost_to_come[node]={'x': x,'y': y,'parent node': "N/A", '
    cost to come':-1.0}
121                     else:
122                         #Obstacle D check
123                         if (x >= 455) and (y >= (105/60)*(x - 515) + 125) and (y
     <= -1*(105/60)*(x - 515) + 125):
124                             cost_to_come[node]={'x': x,'y': y,'parent node': "N/A"
    , 'cost to come':-1.0}
125                         else:
126                             #Walls check
127                             if (x <= 5) or (x >= 595) or (y <= 5) or (y >= 245):
128                                 cost_to_come[node]={'x': x,'y': y,'parent node': "N
    /A", 'cost to come':-1.0}
129                             else:
130                                 cost_to_come[node]={'x': x,'y': y,'parent node': "N
    /A", 'cost to come': float('inf')}
131                                 visual_map[y,x,:] = [100,100,100]
132         # node = node + 1
133
134 #Visualize Space
135 cv2.imshow("Zeros matx", visual_map)  # show numpy array
136 cv2.waitKey(0)  # wait for ay key to exit window
137 cv2.destroyAllWindows()  # close all windows
138
139 SolutionFound = False
140 # parent_node=0
141 counter=0
```

```
142
143 #Initialize the starting point
144 start_cost=0.0
145 bad_choice=True
146 while bad_choice == True:
147     start_x=input('What is the x coordinate (integer only) of your starting point?\
    n')
148     start_x=int(start_x)
149     start_y=input('What is the y coordinate (integer only) of your starting point?\
    n')
150     start_y=int(start_y)
151     start_node=(start_x,start_y)
152     if cost_to_come[start_node]['cost to come'] == -1:
153         print('This is an invalid starting position, please try again.\n')
154     else:
155         bad_choice=False
156
157 cost_to_come[start_node]['cost to come']=0.0
158
159 #Define goal node by x,y coordinates
160 bad_choice = True
161 while bad_choice == True:
162     goal_x = input('What is the x coordinate (integer only) of your target position
    ?\n')
163     goal_x=int(goal_x)
164     goal_y = input('What is the y coordinate (integer only) of your target position
    ?\n')
```

```
165     goal_y=int(goal_y)
166     goal_node=(goal_x, goal_y)
167     if cost_to_come[goal_node]['cost to come'] == -1:
168         print('This is an invalid target position, please try again.\n')
169     else:
170         bad_choice = False
171 
172 #Initialize list of nodes that need to be expanded/investigated/have moves applied
173 queue={}
174 queue[start_node]=cost_to_come[start_node]['cost to come']
175 #print(queue)
176 closed_list = []
177 while SolutionFound!=True:           #counter<10:
178     # print("Counter", counter)
179     #sort queue for lowest cost_to_come
180     # cost_order={}
181 
182     queue=sorted(queue.items(), key = lambda cost: cost[1])
183 
184     # identify the parent node and eliminate it from the list of nodes that need to
     be investigated
185     parent_node=queue.pop(0)
186     parent_node=parent_node[0]
187     queue=dict(queue)
188     # check if the last popped node is a match for goal
189     # if it's a match, initializes reverse_path list and adds node to node map
190 
```

```
191     SolutionFound=check_for_goal(parent_node, goal_node)
192     if SolutionFound == True:
193         print(cost_to_come[parent_node])
194         reverse_path = [goal_node,cost_to_come[goal_node]['parent node']]
195         print(reverse_path)
196         break
197 
198     closed_list.append(parent_node)
199 
200 # #     #perform "moves" on "parent" node to create "new" nodes
201 # #     #each function checks for a valid move, if not valid, returns "parent" node
202 
203     up=move_up(cost_to_come,parent_node)
204     # print(cost_to_come[parent_node])
205     # print(cost_to_come[up])
206 
207     up_right=move_up_right(cost_to_come,parent_node)
208     # print(cost_to_come[parent_node])
209     # print(cost_to_come[up_right])
210     right=move_right(cost_to_come,parent_node)
211     # print(cost_to_come[parent_node])
212     # print(cost_to_come[right])
213     down_right=move_down_right(cost_to_come,parent_node)
214     # print(cost_to_come[parent_node])
215     # print(cost_to_come[down_right])
216     down=move_down(cost_to_come,parent_node)
217     # print(cost_to_come[parent_node])
```

```
218     # print(cost_to_come[down])
219     down_left=move_down_left(cost_to_come,parent_node)
220     # print(cost_to_come[parent_node])
221     # print(cost_to_come[down_left])
222     left=move_left(cost_to_come,parent_node)
223     # print(cost_to_come[parent_node])
224     # print(cost_to_come[left])
225     up_left=move_up_left(cost_to_come,parent_node)
226     # print(cost_to_come[parent_node])
227     # print(cost_to_come[up_left])
228 
229     #stores the "new" nodes in another dictionary for future use in loop
230     action_dict={0:up, 1:up_right,2:right,3:down_right,4:down,5:down_left,6:left, 7
    : up_left}
231 
232     #initialize variable before loop begins
233     match=False
234 
235     #for each action in action_dict loop runs (outer for-loop)
236     for j in range(len(action_dict)):
237         match=False
238 #         #inner for-loop checks one "new" node from action_dict to determine
239         for i in range(len(closed_list) - 1, -1, -1):
240             # print("length closed list =", + len(closed_list))
241             #checks if "new" node is already in the closed list
242             if action_dict[j][0] == closed_list[i] or cost_to_come[action_dict[j][0
    ]]['cost to come'] == -1:
```

```
243                 i=i+1
244                 break
245             #checks if "new" node is in an obstable space
246             # if cost_to_come[action_dict[j][0]]['cost to come'] == -1:
247             #     i=i+1
248             #     break
249             #if node is not in an obstacle or in the closed list, calc new cost-to-
    come
250             # print(cost_to_come[parent_node])
251             new_cost_to_come = cost_to_come[parent_node]['cost to come'] +
    action_dict[j][1]
252             if cost_to_come[parent_node]['cost to come'] + action_dict[j][1] <
    cost_to_come[action_dict[j][0]]['cost to come']:
253                 cost_to_come[action_dict[j][0]]['cost to come']=cost_to_come[
    parent_node]['cost to come'] + action_dict[j][1]
254                 cost_to_come[action_dict[j][0]]['parent node']=parent_node
255                 node=action_dict[j][0]
256                 queue[node]=cost_to_come[action_dict[j][0]]['cost to come']
257     counter=counter+1
258 #    print(counter)
259 #This loop creates the reverse path by searching for the next parent node until the
     start node's parent "NA" is found
260 forward_path=generate_path(reverse_path)
261
262 cv2.imshow("Zeros matx", visual_map)  # show numpy array
263 cv2.waitKey(0)  # wait for ay key to exit window
264 cv2.destroyAllWindows()  # close all windows
```

```
265
266
267 x=[]
268 y=[]
269 visual_map_explore=visual_map
270 for i in range(len(closed_list)):
271     coord=closed_list[i]
272     x=250-int(coord[1])
273     y=int(coord[0])
274     visual_map_explore[x][y]=0
275     cv2.imshow("Zeros matx", visual_map_explore)  # show numpy array
276     cv2.waitKey(1)  # wait for ay key to exit window
277
278 cv2.destroyAllWindows()  # close all windows
279
280 x=[]
281 y=[]
282 for i in range(len(forward_path)):
283     coord=forward_path[i]
284     x=250-int(coord[1])
285     y=int(coord[0])
286     visual_map[x,y,:] = [0,255,0]
287     cv2.imshow("Zeros matx", visual_map)  # show numpy array
288     cv2.waitKey(50)  # wait for ay key to exit window
289
290 cv2.waitKey(0)  # wait for ay key to exit window
291
```

```
292 cv2.destroyAllWindows()  # close all windows
293
294
```