**A\* Algorithm**

**Implementation**

**Name :** G Charan Sri Sai Shankar

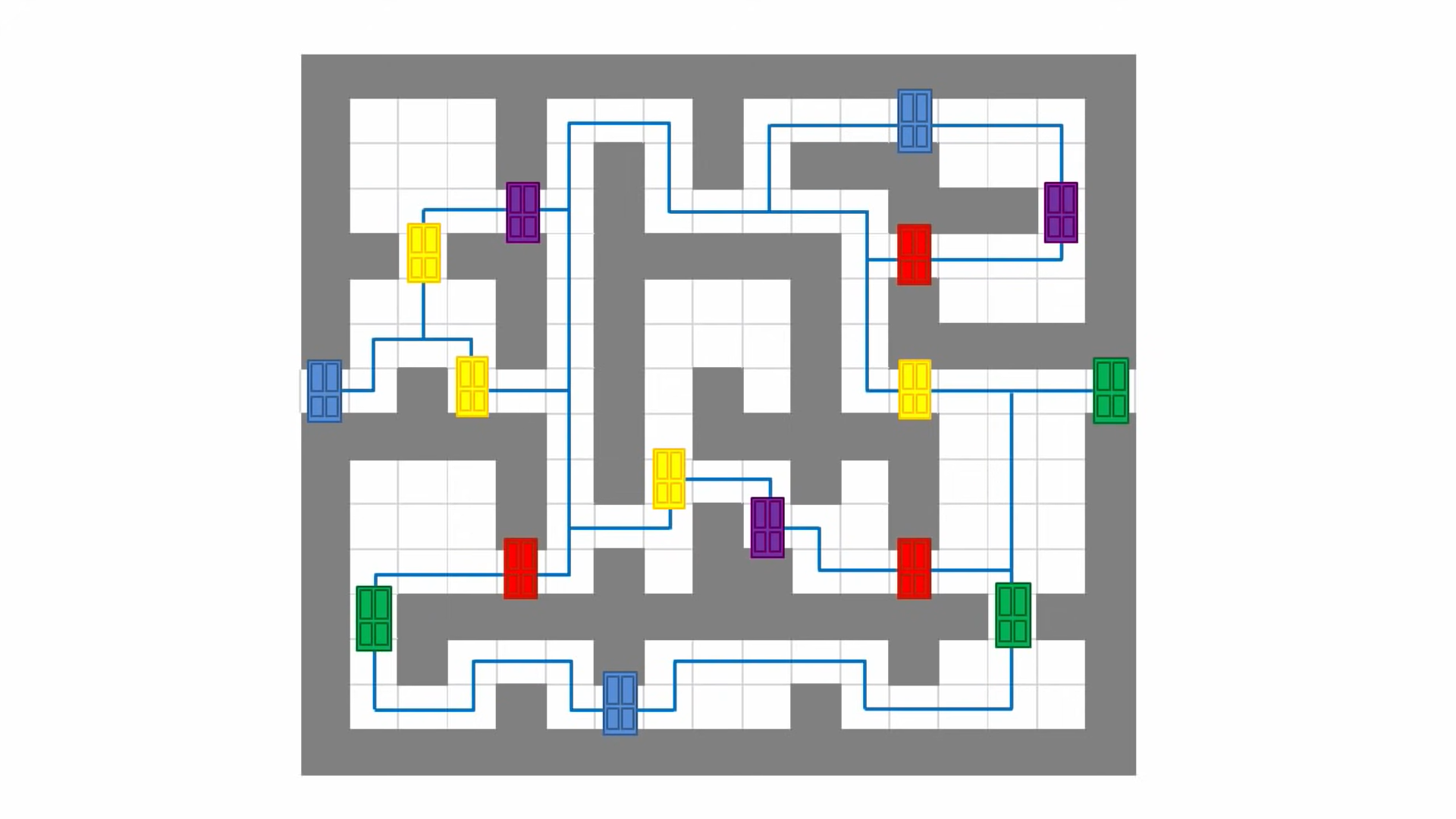
**Roll No :** 411922

**Faculty : Dr. Hima Bindhu Mam**

**Problem statement :**

**Takeshi’s Solver :**

We have a Takeshi’s castle in our city and we have the complete upper view of the castle. The doors , The Walls are clearly seen.

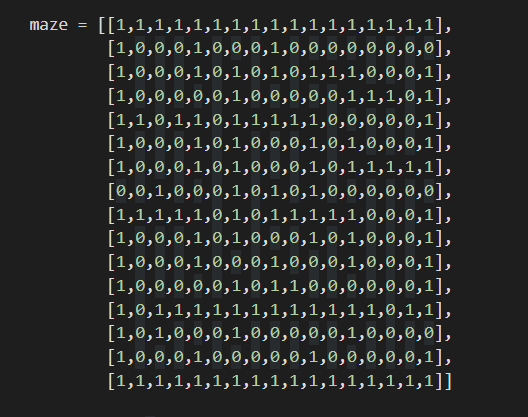


We have to find the Final door where Gold is hidden. Here we have to find the shortest path for the Gold.

We will solve this using A\* Search Algorithm.

**Solution :**

We will give the above castle as the following input.



Paths =0

Walls = 1

f(n)=g(n)+h(n)

Here we take the heuristic function h(n) as

Euclidean distance = (p-q)^2 + (r-s)^2

**Code :**

import numpy as np

class Node:

def \_\_init\_\_(self, parent=None, position=None):

self.parent = parent

self.position = position

self.g = 0

self.h = 0

self.f = 0

def \_\_eq\_\_(self, other):

return self.position == other.position

"""

parent is parent of the current Node

position is current position of the Node in the castle

g is cost from start to current Node

h is heuristic based estimated cost

f is total cost of present node , f = g + h

"""

#This function return the path of the search

def return\_path(current\_node,maze):

path = []

no\_rows, no\_columns = np.shape(maze)

# here we create the initialized result maze with -1 in every position

result = [[-1 for i in range(no\_columns)] for j in range(no\_rows)]

current = current\_node

while current is not None:

path.append(current.position)

current = current.parent

# Return reversed path as we need to show from start to end path

path = path[::-1]

start\_value = 0

# we update the path of start to end found by A-star serch with every step incremented by 1

for i in range(len(path)):

result[path[i][0]][path[i][1]] = start\_value

start\_value += 1

return result

def search(maze, cost, start, end):

# Create start and end node with initized values for g, h and f

start\_node = Node(None, tuple(start))

start\_node.g = start\_node.h = start\_node.f = 0

end\_node = Node(None, tuple(end))

end\_node.g = end\_node.h = end\_node.f = 0

# Initialize both yet\_to\_visit and visited list

# in this list we will put all node that are yet\_to\_visit for exploration.

# From here we will find the lowest cost node to expand next

yet\_to\_visit\_list = []

# in this list we will put all node those already explored so that we don't explore it again

visited\_list = []

# Add the start node

yet\_to\_visit\_list.append(start\_node)

# Adding a stop condition. This is to avoid any infinite loop and stop

# execution after some reasonable number of steps

outer\_iterations = 0

max\_iterations = (len(maze) // 2) \*\* 10

# what squares do we search . serarch movement is left-right-top-bottom

#(4 movements) from every positon

move = [[-1, 0 ], # go up

[ 0, -1], # go left

[ 1, 0 ], # go down

[ 0, 1 ]] # go right

#find maze has got how many rows and columns

no\_rows, no\_columns = np.shape(maze)

# Loop until you find the end

while len(yet\_to\_visit\_list) > 0:

# Every time any node is referred from yet\_to\_visit list, counter of limit operation incremented

outer\_iterations += 1

# Get the current node

current\_node = yet\_to\_visit\_list[0]

current\_index = 0

for index, item in enumerate(yet\_to\_visit\_list):

if item.f < current\_node.f:

current\_node = item

current\_index = index

# if we hit this point return the path such as it may be no solution or

# computation cost is too high

if outer\_iterations > max\_iterations:

print ("giving up on pathfinding too many iterations")

return return\_path(current\_node,maze)

# Pop current node out off yet\_to\_visit list, add to visited list

yet\_to\_visit\_list.pop(current\_index)

visited\_list.append(current\_node)

# test if goal is reached or not, if yes then return the path

if current\_node == end\_node:

return return\_path(current\_node,maze)

# Generate children from all adjacent squares

children = []

for new\_position in move:

# Get node position

node\_position = (current\_node.position[0] + new\_position[0], current\_node.position[1] + new\_position[1])

# Make sure within range (check if within maze boundary)

if (node\_position[0] > (no\_rows - 1) or

node\_position[0] < 0 or

node\_position[1] > (no\_columns -1) or

node\_position[1] < 0):

continue

# Make sure walkable terrain

if maze[node\_position[0]][node\_position[1]] != 0:

continue

# Create new node

new\_node = Node(current\_node, node\_position)

# Append

children.append(new\_node)

# Loop through children

for child in children:

# Child is on the visited list (search entire visited list)

if len([visited\_child for visited\_child in visited\_list if visited\_child == child]) > 0:

continue

# Create the f, g, and h values

child.g = current\_node.g + cost

## Heuristic costs calculated here, this is using eucledian distance

child.h = (((child.position[0] - end\_node.position[0]) \*\* 2) +

((child.position[1] - end\_node.position[1]) \*\* 2))

child.f = child.g + child.h

# Child is already in the yet\_to\_visit list and g cost is already lower

if len([i for i in yet\_to\_visit\_list if child == i and child.g > i.g]) > 0:

continue

# Add the child to the yet\_to\_visit list

yet\_to\_visit\_list.append(child)

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

maze = [[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],

[1,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0],

[1,0,0,0,1,0,1,0,1,0,1,1,1,0,0,0,1],

[1,0,0,0,0,0,1,0,0,0,0,0,1,1,1,0,1],

[1,1,0,1,1,0,1,1,1,1,1,0,0,0,0,0,1],

[1,0,0,0,1,0,1,0,0,0,1,0,1,0,0,0,1],

[1,0,0,0,1,0,1,0,0,0,1,0,1,1,1,1,1],

[0,0,1,0,0,0,1,0,1,0,1,0,0,0,0,0,0],

[1,1,1,1,1,0,1,0,1,1,1,1,1,0,0,0,1],

[1,0,0,0,1,0,1,0,0,0,1,0,1,0,0,0,1],

[1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1],

[1,0,0,0,0,0,1,0,1,1,0,0,0,0,0,0,1],

[1,0,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1],

[1,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,0],

[1,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,1],

[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]]

start = [7, 0] # starting position

end = [7,16] # ending position

cost = 1 # cost per movement

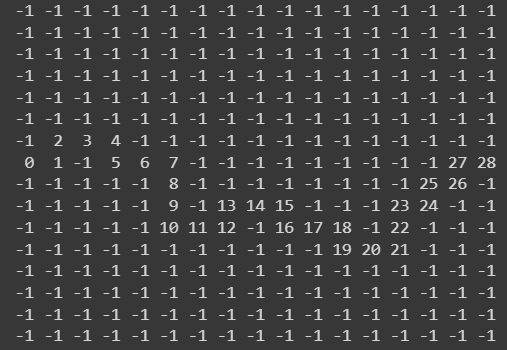
path = search(maze,cost, start, end)

print(path)

print('\n'.join([''.join(["{:" ">3d}".format(item) for item in row])

for row in path]))

**Output :**

****

**Problems i faced :**

Understanding how the Algorithm works is easy but implementing the code is hard. I used many online references for this A\* Implementation. Some of them are

1. The coding train
2. Sebastian lague
3. Tech with Tim
4. Computer science

These are the main sources I referred to.