Assignment 3 Part A

Vasu Bansal, Roll No. 160776, Course - ME766

Path planning in uniform grid-space with obstacles.
Algorithm used - A Star
Assignment is written in *Python* and implemented on *Jupyter Notebook*

In [1]:
```
# Import necessary libraries
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.ticker as plticker
import math
import cv2
import copy
```

In [2]:
```
# To show image in new window
%matplotlib qt
```

In [3]:
```
# Define starting and goal positions
startX, startY = 55,55
goalX, goalY =  90,90

# Workspace Figure settings
fig = plt.figure()
ax = fig.add_subplot(1,1,1)

# Loading Workspace Image
workspace = cv2.imread('workspace_1.png',0)
workspace= workspace[::,::]

# Thresholding the image to remove possible noisy cells
_, workspace = cv2.threshold(workspace, 127, 255, cv2.THRESH_BINARY)

workspaceplot = ax.imshow(workspace)

ax.scatter(startX, startY,marker='x',color='red')
ax.scatter(goalX, goalY, marker='o', color='blue')
ax.axis([0, 100, 0, 100])

# Grid the workspace
major_ticks = np.arange(0, 100, 1)
ax.set_xticks(major_ticks)
ax.set_yticks(major_ticks)
ax.grid('on')
```

```
C:\ProgramData\Anaconda3\lib\site-packages\matplotlib\cbook\__init__.py:424: Mat
plotlibDeprecationWarning:
Passing one of 'on', 'true', 'off', 'false' as a boolean is deprecated; use an a
ctual boolean (True/False) instead.
  warn_deprecated("2.2", "Passing one of 'on', 'true', 'off', 'false' as a "
```

The function retrace_path(currentNode, closedList) retraces the path given the destination node(reached by algorithm) and the closed list.

In [4]:
```
def retrace_path(currentNode, closedList):
    path = [(currentNode.x, currentNode.y)]
    temp = copy.copy(currentNode)

    while(not(temp.x==startX and temp.y==startY)):
        a,b = temp.get_parent()
        temp = copy.copy(closedList[(a,b)])
        path.append((a,b))
#         print(a,b, temp.get_g(), temp.h, temp.get_f())
    path.append((temp.x, temp.y))
    return path
```

dist(currentNode, neighbour) gives the distance between two nodes. Can be user defined

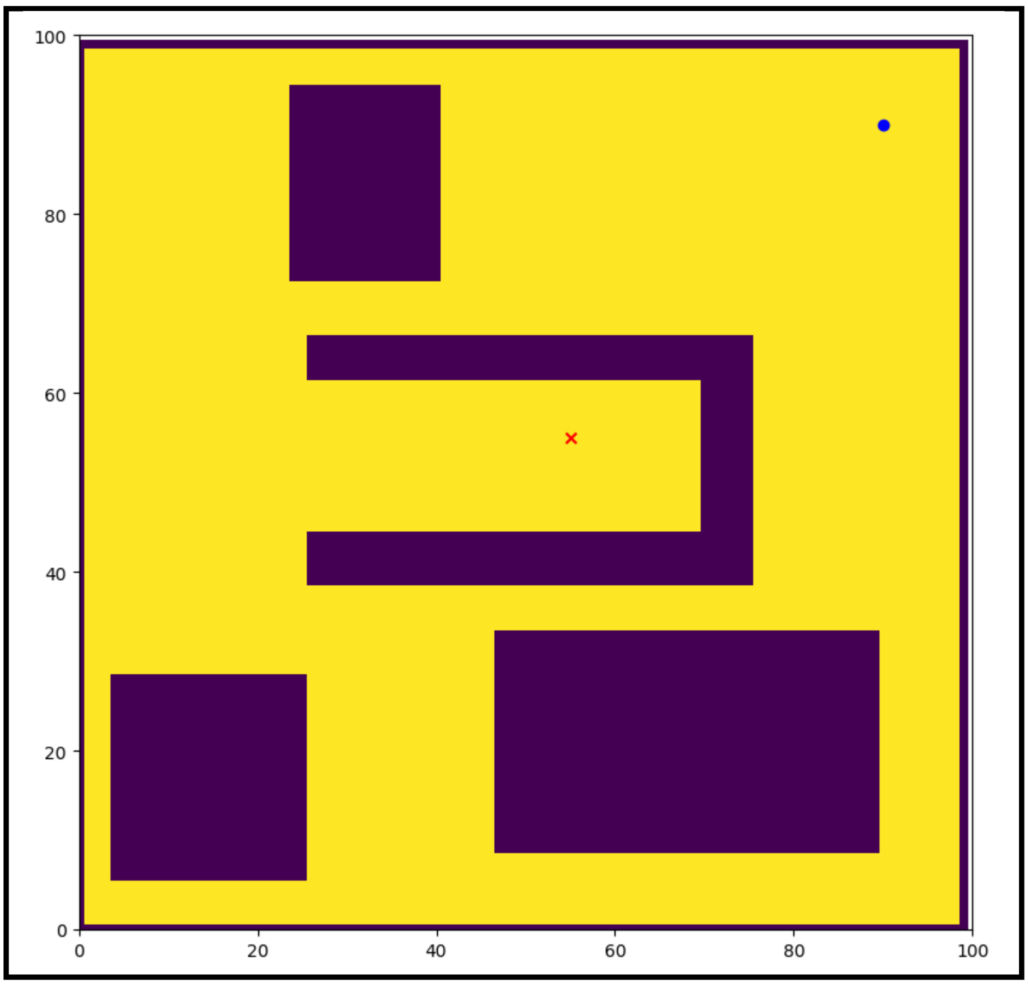
In [5]:
```
def dist(currentNode, neighbour):
    if (not(abs(currentNode.x-neighbour.x)==0 or abs(currentNode.y-neighbour.y)==
0)):
        return 1.4
    else:
        return 1
#     return math.sqrt((currentNode.x-neighbour.x)**2+(currentNode.y-neighbour.y)**
2)
```

Node object has x,y for position.

g - distance from starting position. h - heuristic value. f = g+h

Workspace is an 100 pixel by 100 pixel image created in MS paint. Obstacles are represented by pixel value of zero.

`



Workspace. Obstacles are purple colored. Red cross is the starting position. Blue dot is the goal position

In [6]:

```
# weights for g-value and h-value
w1 = 1
w2 = 1

# Defined node object
class node():
    def __init__(self, xnode, ynode):
        self.x = xnode
        self.y = ynode

        dx = abs(self.x-goalX)
        dy = abs(self.y-goalY)
        self.h = math.sqrt(dx*dx+dy*dy)

        self.g = 999999999999999
        self.f = 999999999999999
        self.parentx=0
        self.parenty=0

    # Methods for node object
    def get_h(self):
        return self.h

    def get_g(self):
        return self.g
    def set_g(self, gval):
        self.g = gval

    def get_f(self):
        return w1*self.g+w2*self.h

    def get_parent(self):
        return self.parentx,self.parenty
    def set_parent(self, parentxval, parentyval):
        self.parentx, self.parenty = parentxval, parentyval

    def isValid(self):
        if(self.x>0 and self.x<100 and self.y>0 and self.y<100):
            return True
        else:
            return False

    def isObstacle(self):   # Tolerance of one cell with obstacle
        if(workspace[self.y,self.x]==0 or workspace[self.y-1,self.x]==0 or workspac
e[self.y,self.x-1]==0 or workspace[self.y-1,self.x-1]==0 or
           workspace[self.y+1,self.x]==0 or workspace[self.y,self.x+1]==0 or worksp
ace[self.y+1,self.x+1]==0 or workspace[self.y+1,self.x-1]==0 or workspace[self.y-1,
self.x+1]==0):
            return True
        else:
            return False

    def isDestination(self):
        if(self.x==goalX and self.y==goalY):
            return True
        else:
            return False
```

In [7]:

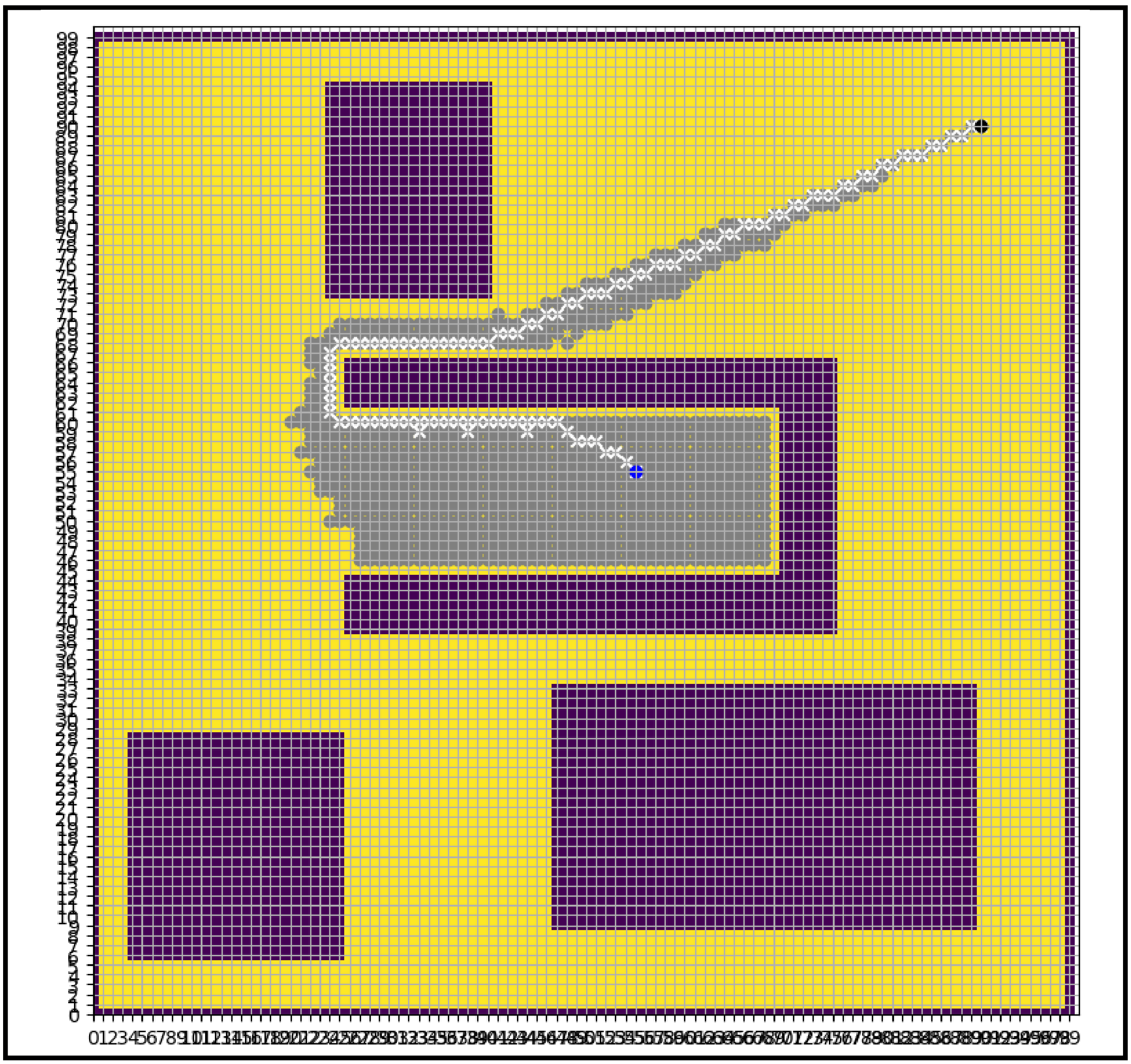
```
def algorithm():

    # Initial check
    if(not node(startX, startY).isValid()):
        print('Starting position is not valid!')
        return None;
    if(not node(goalX, goalY).isValid()):
        print('Goal position is not valid!')
        return None;
    if(node(startX, startY).isObstacle()):
        print('Starting position is in obstacle!')
        return None;
    if(node(goalX, goalY).isObstacle()):
        print('Goal position is in obstacle!')
        return None;

    # Initialised stating node
    startingNode = copy.copy(node(startX, startY))
    startingNode.set_parent(startingNode.x, startingNode.y)
    startingNode.set_g(0)

    # Initialised open and closed lists
    openList = {} # Contains nodes, which have been visited but whose neighbours ar
e not considered
    openList = {(startingNode.x, startingNode.y):startingNode}
    closedList = {} # Contains nodes whhose neighbours are expanded

    while(not(len(openList)==0)):
        # Find the node which has minimum f-value in the open list
        firstKey = list(openList)[0]
        currentNode = copy.copy(openList[firstKey])

        for node_ in openList.values():
            if(currentNode.get_f()>node_.get_f()):
                currentNode = copy.copy(node_)

        print('.',end='')
        ax.scatter(currentNode.x,currentNode.y,color='grey',marker='o')

        # If this node is our destination, we are done.
        if(currentNode.isDestination()):
            print('Destination is Found!')
            path = retrace_path(copy.copy(currentNode), closedList)
            return currentNode, path
        else:

            #Generate neighbors of current node
            neighbours_of_current_node = [node(currentNode.x-1,currentNode.y),node
(currentNode.x,currentNode.y-1),
                                          node(currentNode.x-1,currentNode.y-1),nod
e(currentNode.x+1,currentNode.y),
                                          node(currentNode.x,currentNode.y+1),node
(currentNode.x+1,currentNode.y+1),
                                          node(currentNode.x+1,currentNode.y-1),nod
e(currentNode.x-1,currentNode.y+1)]

            currentNode = copy.copy(currentNode)
            # Delete currentNode from the openlist and add it to closedlist
            del openList[(currentNode.x,currentNode.y)]
            closedList[(currentNode.x,currentNode.y)] = currentNode

            for neighbour in neighbours_of_current_node:

                # Checking if our neighbour is valid and not an obstacle
```

In [8]:
```python
__, path = algorithm()
```

```
.................................................................................
.................................................................................
.................................................................................
.................................................................................
.................................................................................
.................................................................................
.................................................................................
.................................................................................
.................................................................................
.................................................................................
.................................................................................
..........................Destination is Found!
```

In [9]:
```python
for pointer in path:
    ax.scatter(pointer[0],pointer[1],marker='x',color='white')
ax.scatter(startX,startY, marker='o', color='blue')
ax.scatter(goalX,goalY, marker='o', color='black')
```

Out[9]: <matplotlib.collections.PathCollection at 0x18d5ec3ee80>



The grey pixels are the nodes in the closed list. The white colored nodes show the path