Applications of Artificial Intelligence

EAI 6010, CRN 70749

Professor Vladimir Shapiro

Module 5: Assignment Week 5 - Robotic AI Applications

Submitted By - Richa Umesh Rambhia

Robotic AI Applications - Mini Project

Implementation of Wavefront Planner and A Star Algorithm

Table of Contents

Introduction

Robots that move by self-control are in demand in the market industry and robobit fields which can perform different jobs and tasks. There are various methods that can be implmented to plan a movement of the track from the map for the robot navigation such as the **Dijkstra algorithm, A* star algorithm, breadth-first search (BFS), depth-first search (DFS), wavefront-based planners, and rapidly exploring random trees.** [3] But when considering the fastest and simplest algorithms that can be used for execution of the robot navigation are *wavefront algorithm and A_star algorithm.*

Wavefront Algorithm is a search algorithm which is used in the robot navigation for field path planner. [1] This algorithm is used to move the robot between the two given points without having any obstacles in between the movement. The wavefront alogrithm uses a breadth first seach algorithm at the destination point. [2] The instructions and movement of the robot basically depends on the wavefront algorithm and the search algorithms that are implemented. The stragtegy of the algorithm is based on the spreading of the wavefronts which could encrypt the distance from the current location of the robot to any given point in the environment until it reaches the target location. [3]

In this project, the aim is to execute the wavefront planner algorithm and a search algorithm to find the shortest path from a given point, and visualize the calculated path of the robot navigation.

Analysis

The project objective is to implement a wavefront planner algorithm or a search algorithm for the robot navigation from a given point to the destination point. Considering the various applications where the autonomous robots are required, they can perform several tasks and thus require to travel freely in a static or a dynamic environment. In order to make machines intelligent, AI knowledge and technology is considered where path planning is considered the most common problem for robot navigation, where the goal is that the robots have to move from starting position to the target position.

In this task, Python is used which will help to find the shortest path and visualize the calculated path using the 4-connectivity and 8-connectivity.

Q1: Using Python, create a script to find the shortest path from point S (start) to point G (goal) through a discretized workspace (6 rows by 12 columns) such as this (“.” designates a cell of free space, “X” represents a cell fully occupied by an obstacle). Use 4-connectivity (research the subject of connectivity):

A1.

Wavefront Algorithm

In [46]:
```
# Importing Libraries

import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
import random
```

Implementation of the Wavefront Algorithm [5]

In [25]:
```
def DistanceFind(z,y,x, k, j, i):
    dist = math.sqrt((i - x)**2 + (j - y)**2 + (k - z)**2)
    return dist
```

In [26]:
```
def addingObstacle1(x,y,z,radius):
    i = 0
    j = 0
    k = 0

    for i in range(RangeX[1]):
        for j in range(RangeY[1]):
            for k in range(RangeZ[1]):
                distance = DistanceFind(z,y,x, k, j, i)
                if distance<radius+1:
                    array2d[k][j][i] = 1
    return
```

In [27]:
```
def addingObstacle2(x,y,z,side):
    #side = side*1.3
    i = 0
    j = 0
    k = 0
    startx = x-side/2
    endx = x+side/2
    starty = y-side/2
    endy = y+side/2
    startz = z-side/2
    endz = z+side/2

    for i in range(RangeX[1]):
        for j in range(RangeY[1]):
            for k in range(RangeZ[1]):
                if j <= endy and j >= starty and i >= startx and i <= endx and k <= endz and k >= startz:
                    array2d[k][j][i] = 1
    return
```

In [28]:
```
def first_validation(z, y, x):
    if x<RangeX[0] or z<RangeZ[0] or y<RangeY[0] or z >= RangeZ[1] or y >= RangeY[1] or x >= RangeX[1]:
        return False
    else:
        if array2d[z][y][x] == 0:
            return True
        else:
            return False
```

In [29]:
```
def second_validation(z, y, x):
    if RangeX[0]<0 or RangeZ[0]<0 or RangeY[0]<0 or z >= RangeZ[1] or y >= RangeY[1] or x >= RangeX[1]:
        return False
    else:
        if array2d[z][y][x] == 1:
            return False
        else:
            return True
```

In [30]:
```
def roundPoints1(z, y, x):
    value = array2d[z][y][x]
    for i in range(-1, 2):
        for j in range(-1,2):
            for k in range(-1,2):
                if (first_validation(z+k, y + j, i+x)):
                    array2d[z+k][y+j][x+i] = value+ 1
    return
```

In [31]:
```
def Round(num):
    while True:
        for i in range(RangeX[1]):
            for j in range(RangeY[1]):
                for k in range(RangeZ[1]):
                    if array2d[k][j][i] == num:
                        roundPoints1(k, j, i)
        num += 1
        if num>((len(range(RangeX[0], RangeX[1]))*(len(range(RangeY[0], RangeY[1]))))):
            print('No path exist')
            quit()
        if array2d[startPoint[2]][startPoint[1]][startPoint[0]]>0:
            break

    return
```

In [32]:
```
def findDistance1(z, y, x):
    dist = math.sqrt((goalPoint[0] - x)**2 + (goalPoint[1] - y)**2 + (goalPoint[2] - z)**2)
    return dist
```

In [33]:
```python
def pathBuilding():
    print ('Starting poing: (' + str(startPoint[0]) + ':' + str(startPoint[1]) + ':' + str(startPoint[2]) + ')' )
    print ('Ending poing: (' + str(goalPoint[0]) + ':' + str(goalPoint[1]) + ':' + str(goalPoint[2]) + ')' )

    nextPoint = [startPoint[2], startPoint[1], startPoint[0]]
    while True:
        distance1 = 999
        distance2 = 999

        value = array2d[nextPoint[0]][nextPoint[1]][nextPoint[2]]
        x = nextPoint[2]
        y = nextPoint[1]
        z = nextPoint[0]
        for i in range(-1, 2):
            for j in range(-1,2):
                for k in range(-1,2):
                    if (second_validation(k+z, j+y, i + x)):
                        if array2d[k+z][j+y][i+x]<value:
                                distance1 = findDistance1(k+z,j+y, i+x)
                                if distance1 < distance2:
                                    distance2 = distance1
                                    nextPoint = [k+z, j+y,i+x]

        print ('('+ str(nextPoint[2])  + ':' + str(nextPoint[1]) + ':' + str(nextPoint[0]) + ')' )

        if (nextPoint[0] == goalPoint[2] and nextPoint[1] == goalPoint[1] and nextPoint[2] == goalPoint[0]):
            break
```

In [34]:
```python
def CheckStart():
    if goalPoint[2]>= RangeZ[1] or goalPoint[1] >=RangeY[1] or goalPoint[0]>= RangeX[1]:
        print("Goal Point is out of bounds ")
        quit()
    elif startPoint[2]>= RangeZ[1] or startPoint[1] >=RangeY[1] or startPoint[0]>= RangeX[1]:
        print("Start Point is out of bounds ")
        quit()
    else:
        array2d[goalPoint[2]][goalPoint[1]][goalPoint[0]] = 2

    if (array2d[goalPoint[2]][goalPoint[1]][goalPoint[0]] == 1 ):
        print("Goal Point is inside object")
        quit()
    elif (array2d[startPoint[2]][startPoint[1]][startPoint[0]] == 1 ):
        print("Start Point is inside object")
        quit()
```

In [35]:
```
RangeX=[0,60]
RangeY=[0,60]
RangeZ=[0,60]


LoadCube1 = [10, 10, 5, 10]
LoadCube2 = [35, 25, 20, 10]
LoadSphere1 = [23, 35, 25, 5]
LoadSphere2 = [50, 50, 30, 5]

goalPoint = [0,0,0]
startPoint = [57,59,27]

s = (RangeZ[1],RangeY[1],RangeX[1])
array2d = np.zeros((s), dtype=int)

# Adding Obstacles
addingObstacle2(LoadCube1[0],LoadCube1[1],LoadCube1[2],LoadCube1[3])
addingObstacle2(LoadCube2[0],LoadCube2[1],LoadCube2[2],LoadCube2[3])
addingObstacle1(LoadSphere1[0],LoadSphere1[1],LoadSphere1[2],LoadSphere1[3])
addingObstacle1(LoadSphere2[0],LoadSphere2[1],LoadSphere2[2],LoadSphere2[3])


#Check validation of start and end poibt
CheckStart()
#DoWavefront
Round(2)

pathBuilding()
print ('Path Complete!')
```

```
Starting poing: (57:59:27)
Ending poing: (0:0:0)
(56:58:26)
(55:57:25)
(54:56:24)
(53:55:23)
(52:54:22)
(51:53:21)
(50:52:20)
(49:51:19)
(48:50:18)
(47:49:17)
(46:48:16)
(45:47:15)
(44:46:14)
(43:45:13)
(42:44:12)
(41:43:11)
(40:42:10)
(39:41:9)
(38:40:8)
(37:39:7)
(36:38:6)
(35:37:5)
(34:36:4)
(33:35:3)
(32:34:2)
(31:33:1)
(30:32:0)
(29:31:0)
(28:30:0)
(27:29:0)
(26:28:0)
(25:27:0)
(24:26:0)
(23:25:0)
(22:24:0)
(21:23:0)
(20:22:0)
(19:21:1)
(18:20:2)
(17:19:3)
(16:18:4)
(15:17:5)
(14:16:6)
(13:16:7)
(12:16:8)
(11:16:9)
(10:16:10)
(9:15:11)
(8:14:11)
(7:13:11)
(6:12:11)
(5:11:11)
(4:10:10)
(3:9:9)
(2:8:8)
(1:7:7)
(0:6:6)
(0:5:5)
(0:4:4)
(0:3:3)
(0:2:2)
(0:1:1)
(0:0:0)
Path Complete!
```

A star Algorithm

A1. Implementation of search algorithm (A star algorithm) using 4 connectivity. [6]

In [87]:
```
class Node():
    """A node class for A* Pathfinding"""

    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
```

In [88]:
```python
def astar(maze, start, end):
    """Returns a list of tuples as a path from the given start to the given end in the given maze"""

    # Create start and end node
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0

    # Initialize both open and closed list
    open_list = []
    closed_list = []

    # Add the start node
    open_list.append(start_node)

    # Loop until you find the end
    while len(open_list) > 0:

        # Get the current node
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index

        # Pop current off open list, add to closed list
        open_list.pop(current_index)
        closed_list.append(current_node)

        # Found the goal
        if current_node == end_node:
            path = []
            current = current_node
            while current is not None:
                path.append(current.position)
                current = current.parent
            return path[::-1] # Return reversed path

        # Generate children
        children = []
        for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0)]: # Adjacent squares

            # Get node position
            node_position = (current_node.position[0] + new_position[0], current_node.position[1] + new_position[1])

            # Make sure within range
            if node_position[0] > (len(maze) - 1) or node_position[0] < 0 or node_position[1] > (len(maze[len(maze)-1]) -1) or no
                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 closed list
            for closed_child in closed_list:
                if child == closed_child:
                    continue

            # Create the f, g, and h values
            child.g = current_node.g + 1
            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 open list
            for open_node in open_list:
                if child == open_node and child.g > open_node.g:
                    continue

            # Add the child to the open list
            open_list.append(child)
```

In [89]:
```python
def main():

    maze = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

    start = (0, 0)
    end = (5, 9)

    path = astar(maze, start, end)
    print("Path:", path)
    print("Length of path:", len(path))

if __name__ == '__main__':
    main()
```

```
Path: [(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (2, 3), (3, 3), (4, 3), (5, 3), (5, 4), (5, 5), (5, 6), (5, 7), (5, 8), (5, 9)]
Length of path: 15
```

Q2: Find a way to visualize the calculated path, including the path length, with the final state of the wave expansion.

A2. The calculated path and the length of the path is visualized using python packages and libraries to better understand the path of the matrix.

In [96]:
```python
# visualizing the original maze

maze = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

start = (0, 0)
end = (5, 9)

fig, ax = plt.subplots(figsize=(10,6))
for i, j in enumerate(maze):
  for k, l in enumerate(j):
    if l==0:
      plt.plot(k,i,'.',color="red")
    else:
      plt.plot(k,i,'X',color="red")

plt.show()
```

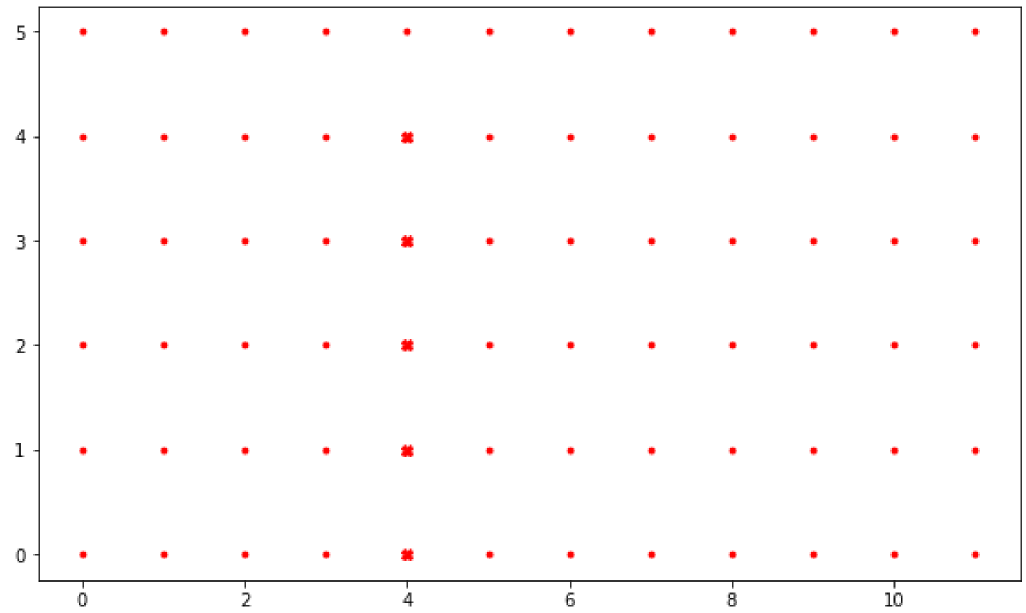


Figure 1: Visualizing the maze of the A star algorithm using 4 connectivity

In [97]:
```python
# visualizing the calculated path of the maze

maze = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

start = (0, 0)
end = (5, 9)

path = astar(maze, start, end)
length_path_1 = len(path)

fig = plt.figure(figsize=(10,6))
az = fig.add_subplot()

for i, j in enumerate(maze):
  for k, l in enumerate(j):
    if l==0:
      az.plot(k,i,'.',color="red")
    else:
      az.plot(k,i,'X',color="red")


coordinates = [(s,r) for (r,s) in path]

x_coordinates = [z for (z, w) in coordinates]
y_coordinates = [w for (z, w) in coordinates]

az.plot(x_coordinates, y_coordinates, '-', color='green')
plt.show()
```

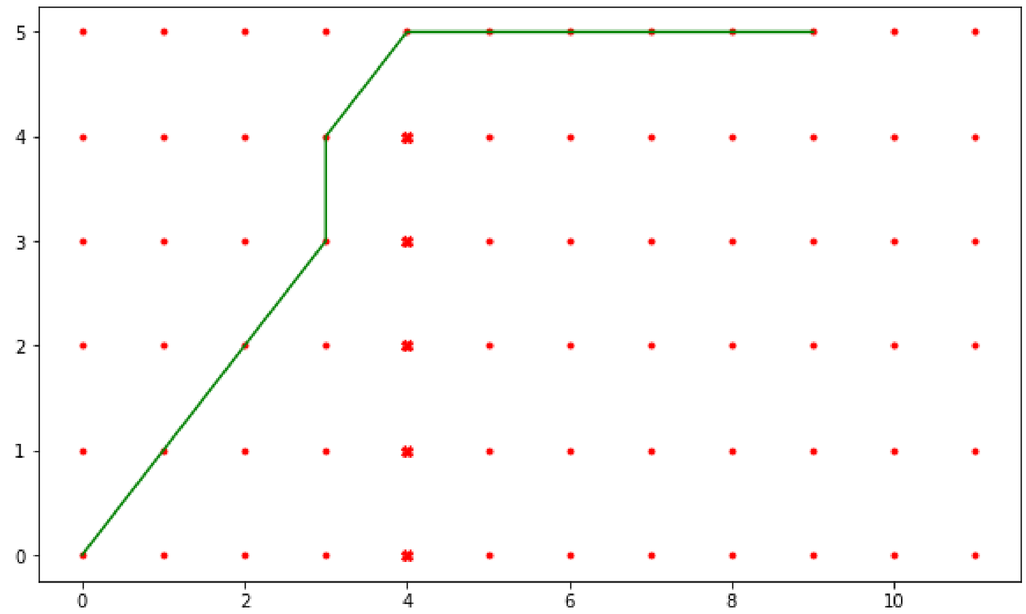


Figure 2: Visualizing the path of maze of the A star algorithm using 4 connectivity

Q3: Change to 8-connectivity, recalculate, visualize and interpret the results.

A3. Implementation of search algorithm (A star algorithm) using 8 connectivity. [6]

In [98]:
```python
class Node():
    """A node class for A* Pathfinding"""

    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
```

In [99]:
```python
def astar(maze, start, end):
    """Returns a list of tuples as a path from the given start to the given end in the given maze"""

    # Create start and end node
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0

    # Initialize both open and closed list
    open_list = []
    closed_list = []

    # Add the start node
    open_list.append(start_node)

    # Loop until you find the end
    while len(open_list) > 0:

        # Get the current node
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index

        # Pop current off open list, add to closed list
        open_list.pop(current_index)
        closed_list.append(current_node)

        # Found the goal
        if current_node == end_node:
            path = []
            current = current_node
            while current is not None:
                path.append(current.position)
                current = current.parent
            return path[::-1] # Return reversed path

        # Generate children
        children = []
        for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]: # Adjacent squares

            # Get node position
            node_position = (current_node.position[0] + new_position[0], current_node.position[1] + new_position[1])

            # Make sure within range
            if node_position[0] > (len(maze) - 1) or node_position[0] < 0 or node_position[1] > (len(maze[len(maze)-1]) -1) or no
                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 closed list
            for closed_child in closed_list:
                if child == closed_child:
                    continue

            # Create the f, g, and h values
            child.g = current_node.g + 1
            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 open list
            for open_node in open_list:
                if child == open_node and child.g > open_node.g:
                    continue

            # Add the child to the open list
            open_list.append(child)
```

In [100]:
```python
def main():

    maze = [[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]]

    start = (0, 0)
    end = (4, 8)

    path = astar(maze, start, end)
    print("Path:", path)
    print("Length of path:", len(path))

if __name__ == '__main__':
    main()
```

```
Path: [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (4, 5), (4, 6), (4, 7), (4, 8)]
Length of path: 9
```

Visualizing the path for A star algorithm using 8 connectivity.

In [102]:
```python
# visualizing the original maze

maze = [[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]]

start = (0, 0)
end = (4, 8)

fig, ax = plt.subplots(figsize=(10,6))
for i, j in enumerate(maze):
  for k, l in enumerate(j):
    if l==0:
      plt.plot(k,i,'.',color="red")

    else:
      plt.plot(k,i,'X',color="red")

plt.show()
```

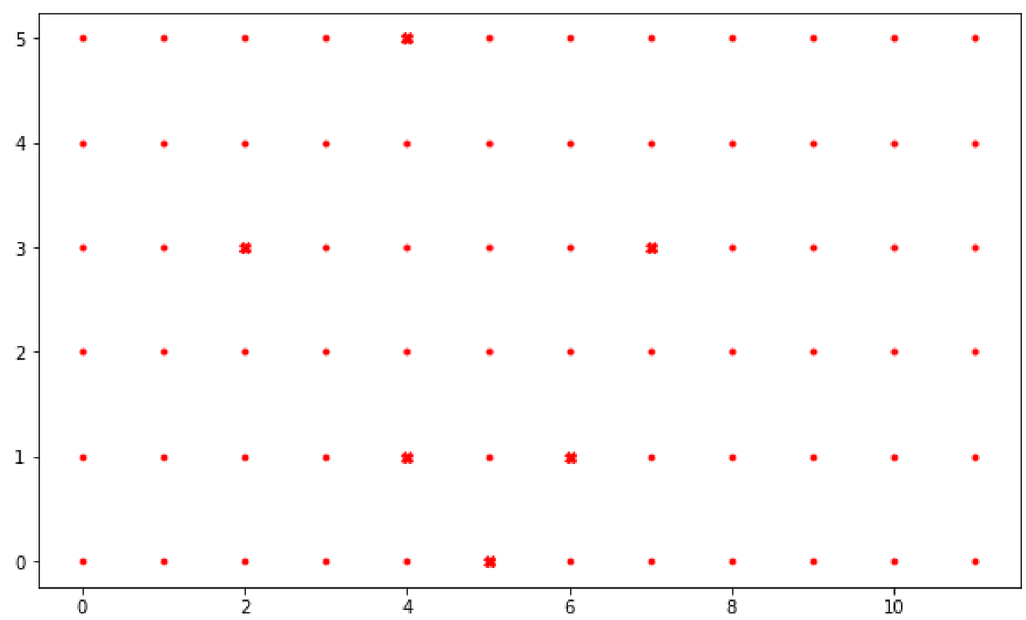


Figure 3: Visualizing the maze of the A star algorithm using 8 connectivity

In [103]:
```python
# visualizing the calculated path of the maze

maze = [[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]]

start = (0, 0)
end = (4, 8)
path = astar(maze, start, end)
length_path_1 = len(path)

fig = plt.figure(figsize=(10,6))
az = fig.add_subplot()

for i, j in enumerate(maze):
  for k, l in enumerate(j):
    if l==0:
      az.plot(k,i,'.',color="red")
    else:
      az.plot(k,i,'X',color="red")


coordinates = [(s,r) for (r,s) in path]

x_coordinates = [z for (z, w) in coordinates]
y_coordinates = [w for (z, w) in coordinates]

az.plot(x_coordinates, y_coordinates, '-', color='green')
plt.show()
```

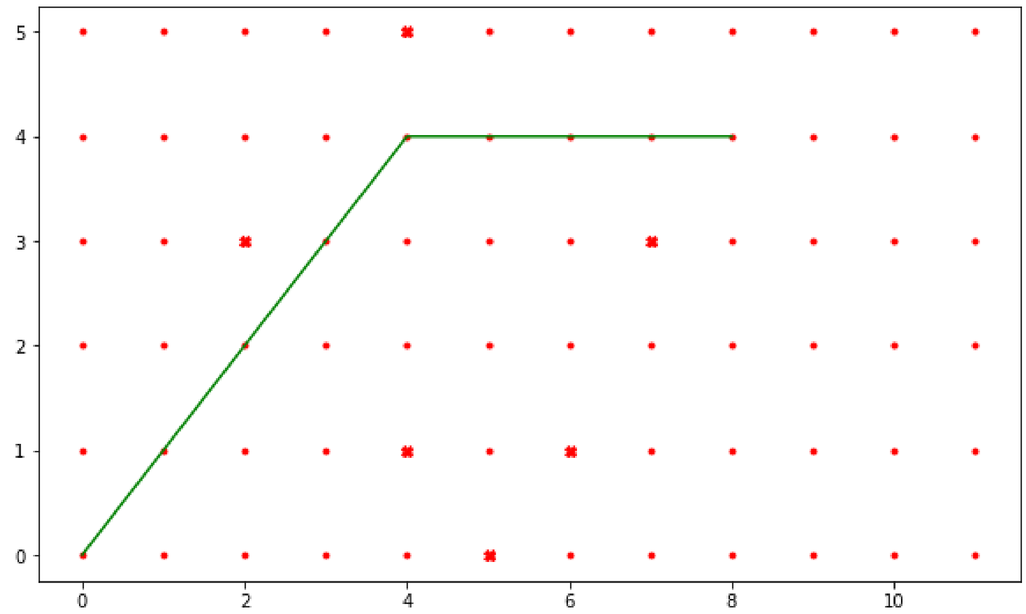


Figure 4: Visualizing the path of maze of the A star algorithm using 8 connectivity

Q4: Modify the code so that the coordinates of “S” and “G” are randomly generated (obviously, the locations of those cannot be in the obstacle area). Run this simulation in a loop. At the end of each simulation, display the path. For the sake of the exercise, repeat the loop just three times. Either connectivity (4 or 8) is acceptable. For simplicity, you can restrict randomly generated “S”s to the left half and “G”s to the right half.

A4. Implementation of A star algorithm using 4 connectivity where the start and end coordinates are randomly generated to determine the path of the maze.

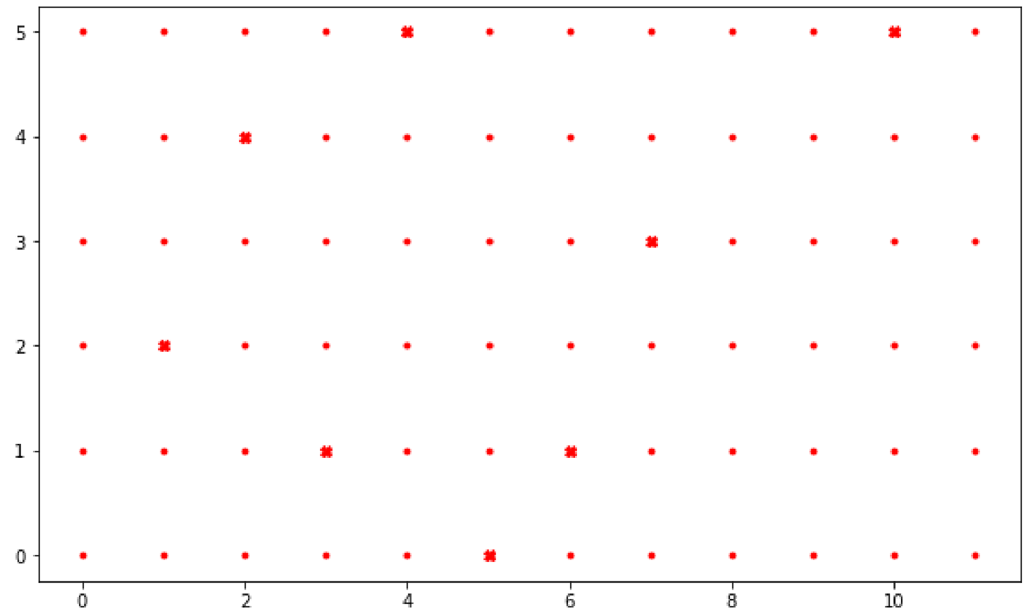
Loop 1 : Implementing the A star algorithm using 4 connectivity where start and end points are randomly generated and visualizing the calculated path

In [118]:
```python
class Node():
    """A node class for A* Pathfinding"""

    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
```

In [119]:
```python
def astar(maze, start, end):
    """Returns a list of tuples as a path from the given start to the given end in the given maze"""

    # Create start and end node
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0

    # Initialize both open and closed list
    open_list = []
    closed_list = []

    # Add the start node
    open_list.append(start_node)

    # Loop until you find the end
    while len(open_list) > 0:

        # Get the current node
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index

        # Pop current off open list, add to closed list
        open_list.pop(current_index)
        closed_list.append(current_node)

        # Found the goal
        if current_node == end_node:
            path = []
            current = current_node
            while current is not None:
                path.append(current.position)
                current = current.parent
            return path[::-1] # Return reversed path

        # Generate children
        children = []
        for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0)]: # Adjacent squares

            # Get node position
            node_position = (current_node.position[0] + new_position[0], current_node.position[1] + new_position[1])

            # Make sure within range
            if node_position[0] > (len(maze) - 1) or node_position[0] < 0 or node_position[1] > (len(maze[len(maze)-1]) -1) or no
                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 closed list
            for closed_child in closed_list:
                if child == closed_child:
                    continue

            # Create the f, g, and h values
            child.g = current_node.g + 1
            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 open list
            for open_node in open_list:
                if child == open_node and child.g > open_node.g:
                    continue

            # Add the child to the open list
            open_list.append(child)
```

In [128]:
```python
def main():

    maze_loop1 = [[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
            [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0]]

    start_loop1 = (random.randint(0,5), random.randint(0,11))
    end_loop1 = (random.randint(0,5), random.randint(0,11))

    path = astar(maze_loop1, start_loop1, end_loop1)
    print("Path:", path)
    print("Length of path:", len(path))
    print("\nVisualization of the original maze - Loop 1\n")

    fig, ax = plt.subplots(figsize=(10,6))
    for i, j in enumerate(maze_loop1):
      for k, l in enumerate(j):
        if l==0:
          plt.plot(k,i,'.',color="red")
        else:
          plt.plot(k,i,'X',color="red")

    plt.show()

    print("\n\nVisualization of the calculated path of the maze - Loop 1\n")
    fig = plt.figure(figsize=(10,6))
    az = fig.add_subplot()

    for i, j in enumerate(maze_loop1):
      for k, l in enumerate(j):
        if l==0:
          az.plot(k,i,'.',color="red")
        else:
          az.plot(k,i,'X',color="red")


    coordinates = [(s,r) for (r,s) in path]

    x_coordinates = [z for (z, w) in coordinates]
    y_coordinates = [w for (z, w) in coordinates]

    az.plot(x_coordinates, y_coordinates, '-', color='green')

if __name__ == '__main__':
    main()
```

```
Path: [(0, 11), (0, 10), (0, 9), (0, 8), (0, 7), (1, 7), (2, 7), (2, 6), (2, 5), (1, 5), (1, 4), (0, 4)]
Length of path: 12

Visualization of the original maze - Loop 1
```




```
Visualization of the calculated path of the maze - Loop 1
```

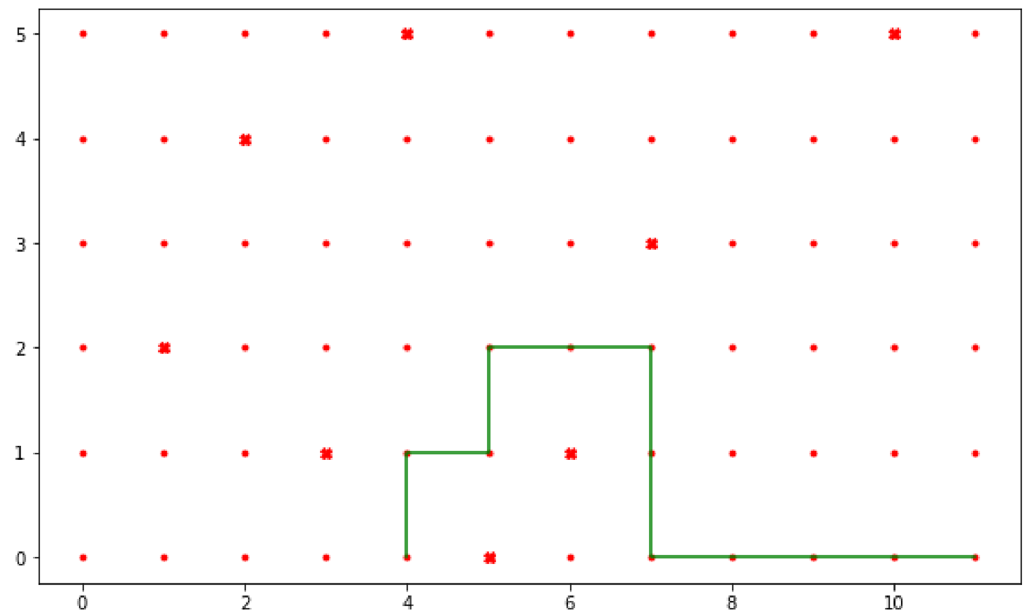


Figure 5: Visualizing the original maze and the path of maze of the A star algorithm using 4 connectivity - Loop 1

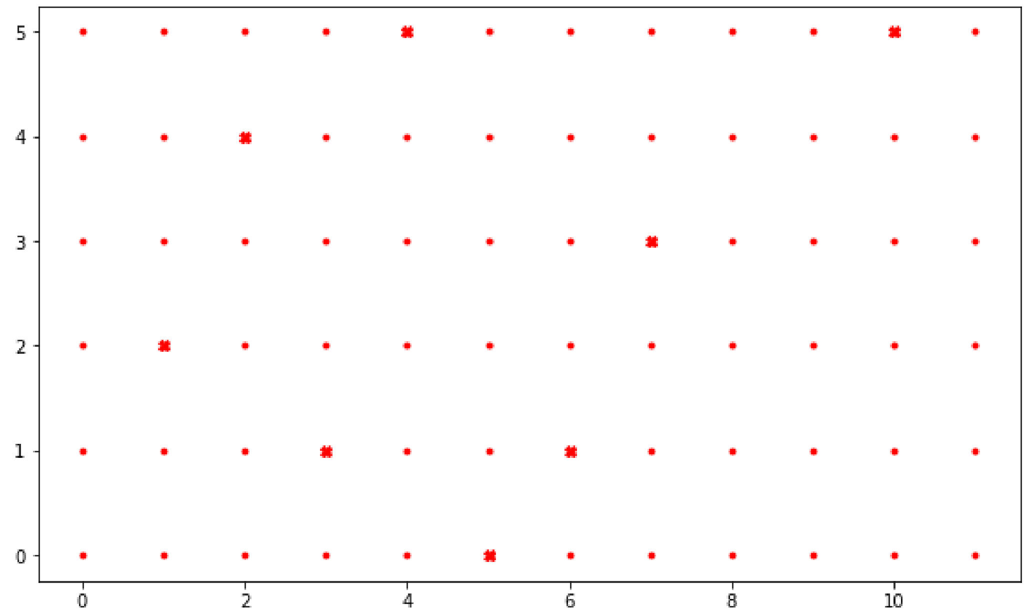
Loop 2 : Implementing the A star algorithm using 4 connectivity where start and end points are randomly generated and visualizing the calculated path

In [121]:
```
class Node():
    """A node class for A* Pathfinding"""

    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
```

In [122]:
```python
def astar(maze, start, end):
    """Returns a list of tuples as a path from the given start to the given end in the given maze"""

    # Create start and end node
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0

    # Initialize both open and closed list
    open_list = []
    closed_list = []

    # Add the start node
    open_list.append(start_node)

    # Loop until you find the end
    while len(open_list) > 0:

        # Get the current node
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index

        # Pop current off open list, add to closed list
        open_list.pop(current_index)
        closed_list.append(current_node)

        # Found the goal
        if current_node == end_node:
            path = []
            current = current_node
            while current is not None:
                path.append(current.position)
                current = current.parent
            return path[::-1] # Return reversed path

        # Generate children
        children = []
        for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0)]: # Adjacent squares

            # Get node position
            node_position = (current_node.position[0] + new_position[0], current_node.position[1] + new_position[1])

            # Make sure within range
            if node_position[0] > (len(maze) - 1) or node_position[0] < 0 or node_position[1] > (len(maze[len(maze)-1]) -1) or no
                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 closed list
            for closed_child in closed_list:
                if child == closed_child:
                    continue

            # Create the f, g, and h values
            child.g = current_node.g + 1
            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 open list
            for open_node in open_list:
                if child == open_node and child.g > open_node.g:
                    continue

            # Add the child to the open list
            open_list.append(child)
```

In [129]:
```python
def main():

    maze_loop2 = [[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
            [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0]]

    start_loop2 = (random.randint(0,5), random.randint(0,11))
    end_loop2 = (random.randint(0,5), random.randint(0,11))

    path = astar(maze_loop2, start_loop2, end_loop2)
    print("Path:", path)
    print("Length of path:", len(path))
    print("\nVisualization of the original maze - Loop 2\n")

    fig, ax = plt.subplots(figsize=(10,6))
    for i, j in enumerate(maze_loop2):
      for k, l in enumerate(j):
        if l==0:
          plt.plot(k,i,'.',color="red")
        else:
          plt.plot(k,i,'X',color="red")

    plt.show()

    print("\n\nVisualization of the calculated path of the maze - Loop 2\n")
    fig = plt.figure(figsize=(10,6))
    az = fig.add_subplot()

    for i, j in enumerate(maze_loop2):
      for k, l in enumerate(j):
        if l==0:
          az.plot(k,i,'.',color="red")
        else:
          az.plot(k,i,'X',color="red")


    coordinates = [(s,r) for (r,s) in path]

    x_coordinates = [z for (z, w) in coordinates]
    y_coordinates = [w for (z, w) in coordinates]

    az.plot(x_coordinates, y_coordinates, '-', color='green')

if __name__ == '__main__':
    main()
```

```
Path: [(4, 10), (3, 10), (3, 9), (2, 9), (2, 8), (1, 8), (1, 7), (0, 7)]
Length of path: 8

Visualization of the original maze - Loop 2
```




```
Visualization of the calculated path of the maze - Loop 2
```

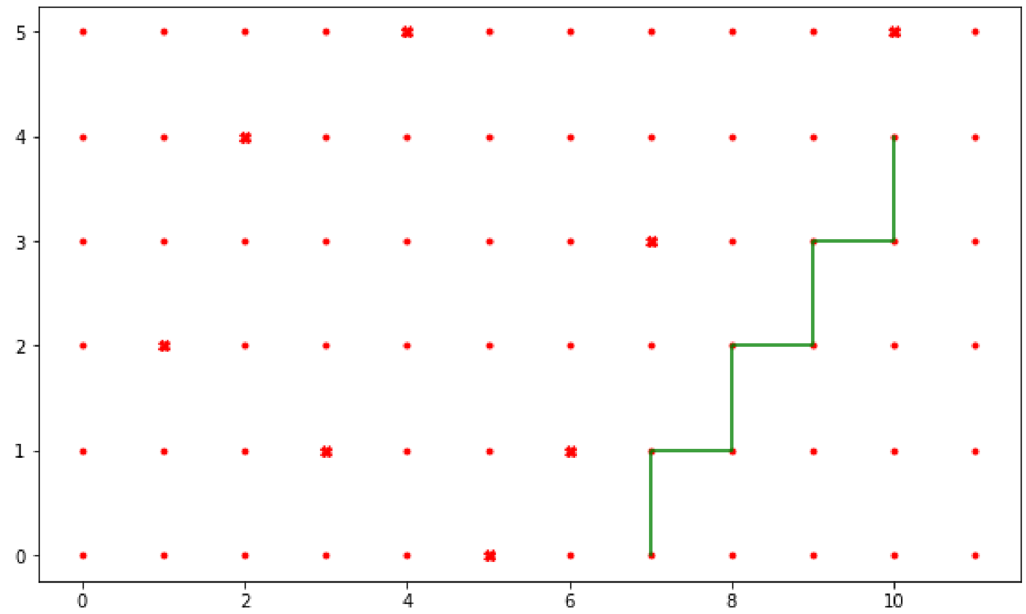


Figure 6: Visualizing the original maze and the path of maze of the A star algorithm using 4 connectivity - Loop 2

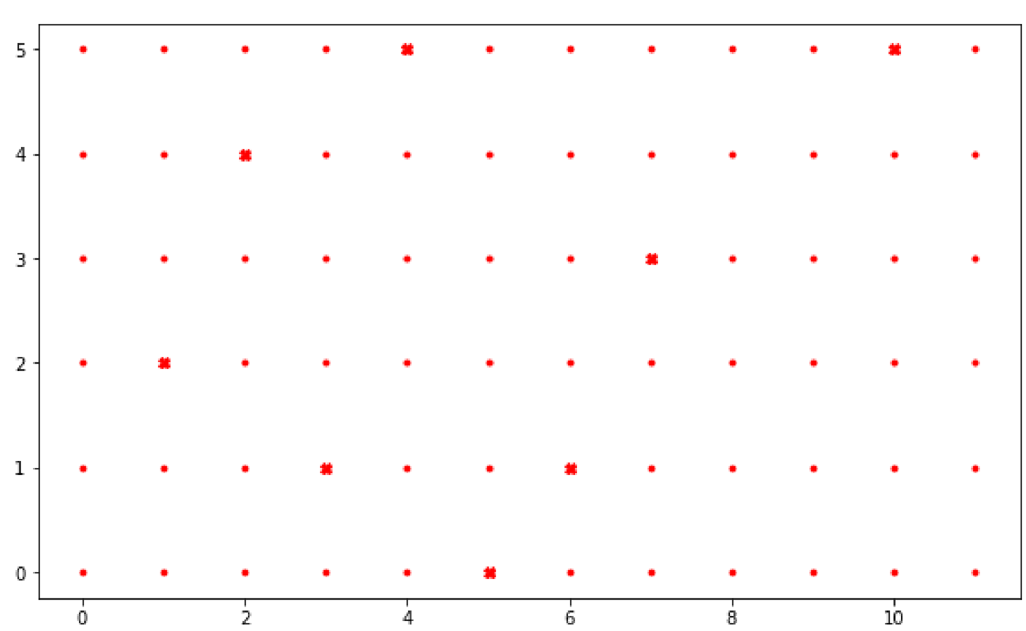
Loop 3 : Implementing the A star algorithm using 4 connectivity where start and end points are randomly generated and visualizing the calculated path

In [125]:
```
class Node():
    """A node class for A* Pathfinding"""

    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
```

In [126]:
```
def astar(maze, start, end):
    """Returns a list of tuples as a path from the given start to the given end in the given maze"""

    # Create start and end node
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0

    # Initialize both open and closed list
    open_list = []
    closed_list = []

    # Add the start node
    open_list.append(start_node)

    # Loop until you find the end
    while len(open_list) > 0:

        # Get the current node
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index

        # Pop current off open list, add to closed list
        open_list.pop(current_index)
        closed_list.append(current_node)

        # Found the goal
        if current_node == end_node:
            path = []
            current = current_node
            while current is not None:
                path.append(current.position)
                current = current.parent
            return path[::-1] # Return reversed path

        # Generate children
        children = []
        for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0)]: # Adjacent squares

            # Get node position
            node_position = (current_node.position[0] + new_position[0], current_node.position[1] + new_position[1])

            # Make sure within range
            if node_position[0] > (len(maze) - 1) or node_position[0] < 0 or node_position[1] > (len(maze[len(maze)-1]) -1) or no
                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 closed list
            for closed_child in closed_list:
                if child == closed_child:
                    continue

            # Create the f, g, and h values
            child.g = current_node.g + 1
            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 open list
            for open_node in open_list:
                if child == open_node and child.g > open_node.g:
                    continue

            # Add the child to the open list
            open_list.append(child)
```

In [132]:
```
def main():

    maze_loop3 = [[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
            [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0]]

    start_loop3 = (random.randint(0,5), random.randint(0,11))
    end_loop3 = (random.randint(0,5), random.randint(0,11))

    path = astar(maze_loop3, start_loop3, end_loop3)
    print("Path:", path)
    print("Length of path:", len(path))
    print("\nVisualization of the original maze - Loop 3\n")

    fig, ax = plt.subplots(figsize=(10,6))
    for i, j in enumerate(maze_loop3):
      for k, l in enumerate(j):
        if l==0:
          plt.plot(k,i,'.',color="red")
        else:
          plt.plot(k,i,'X',color="red")

    plt.show()

    print("\n\nVisualization of the calculated path of the maze - Loop 3\n")
    fig = plt.figure(figsize=(10,6))
    az = fig.add_subplot()

    for i, j in enumerate(maze_loop3):
      for k, l in enumerate(j):
        if l==0:
          az.plot(k,i,'.',color="red")
        else:
          az.plot(k,i,'X',color="red")


    coordinates = [(s,r) for (r,s) in path]

    x_coordinates = [z for (z, w) in coordinates]
    y_coordinates = [w for (z, w) in coordinates]

    az.plot(x_coordinates, y_coordinates, '-', color='green')

if __name__ == '__main__':
    main()
```

```
Path: [(4, 9), (4, 8), (4, 7), (4, 6), (4, 5), (4, 4), (4, 3), (3, 3), (3, 2), (2, 2), (1, 2), (1, 1), (1, 0), (0, 0)]
Length of path: 14

Visualization of the original maze - Loop 3
```




```
Visualization of the calculated path of the maze - Loop 3
```

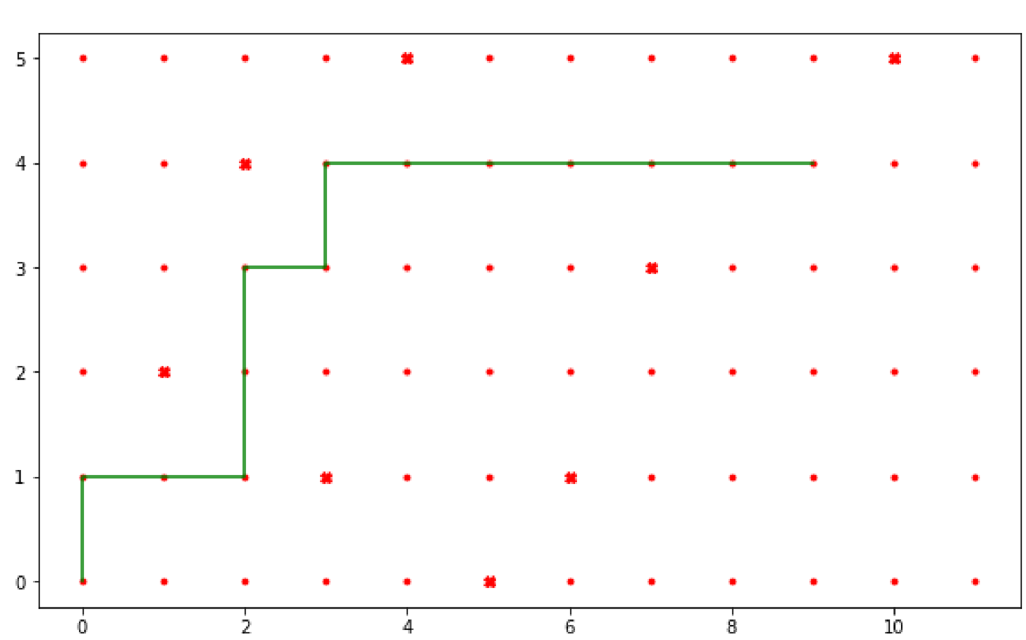


Figure 7: Visualizing the original maze and the path of maze of the A star algorithm using 4 connectivity - Loop 3

Q5: Reflect on the results.

A5. Results:

The implementation of the search algorithm i.e., *Wavefront algorithm and A star algorithm* is executed in order to find the path of the matrix and visualize it. The outputs of the algorithm along with the visualizations help in understanding the path that is obtained through these algorithms where the **4 connectivity and 8 connectivity path** is implemented.

The algorithm is implemented for 4 connectivity where the start and end points of the maze are provided to the algorithm. Similarly, for 8 connectivity, the start and ends points are provided where the calculated path is visualized. In the last task, using 4 connectivity, the start and end coordinates are *randomly generated using the python function* and the calculated path along with the length of the path is displayed and visualized. This process is iterated in three loops where the output of each loop is visualized and we observe the path obtained that is visualized in these three loops.

Q6: Run all the cells, ensure all are executed and create the output.

A6. All the cells were executed and the output was obtained.

Conclusion

The implementation of the search algorithms such as Wavefront algorithm and A star algorithm help in **determining the path of the matrix or the maze** which is done using the optimal and efficient path finding algorithms. Thus, in this assignment the aim was to implement the search algorithm in order to find the optimal path of the maze and visualize it in order to better understand the path that was found using the search algorithm for the maze.

A star algorithm is implemented for 4 connectivity and 8 connectivity matrix where for the first task the start and end points of the maze are predefined, whereas for the second task the coordinates are randomly generated by the function and we obtained different paths for these loops that are implemented.

Hence, the search algorithms for optimal path finding is implemented using A star algorithm.

References

[1] How to Build a Robot Tutorials - Society of Robots. (n.d.). https://www.societyofrobots.com/programming_wavefront.shtml (https://www.societyofrobots.com/programming_wavefront.shtml)

[2] 16-311 Lab 5: Motion Planning (Wavefront Algorithm). (n.d.). https://www.cs.cmu.edu/%7E16311/s07/labs/lab05/ (https://www.cs.cmu.edu/%7E16311/s07/labs/lab05/)

[3] Zidane, I. M., & Ibrahim, K. (2017). Wavefront and A-Star Algorithms for Mobile Robot Path Planning. Proceedings of the International Conference on Advanced Intelligent Systems and Informatics 2017, 69–80. https://doi.org/10.1007/978-3-319-64861-3_7 (https://doi.org/10.1007/978-3-319-64861-3_7)

[4] Gul, F., Rahiman, W., & Nazli Alhady, S. S. (2019). A comprehensive study for robot navigation techniques. Cogent Engineering, 6(1). https://doi.org/10.1080/23311916.2019.1632046 (https://doi.org/10.1080/23311916.2019.1632046)

[5] Madden, D. (n.d.). GitHub - dmitriimadden/pathfinding at 352d1959f107b2318d7fae52685b5a58fe5a9602. GitHub. https://github.com/dmitriimadden/pathfinding/tree/352d1959f107b2318d7fae52685b5a58fe5a9602 (https://github.com/dmitriimadden/pathfinding/tree/352d1959f107b2318d7fae52685b5a58fe5a9602)

[6] Swift, N. (2020, May 30). Easy A* (star) Pathfinding - Nicholas Swift. Medium. https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2 (https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2)