CMR INSTITUTE OF TECHNOLOGY

18CSL76

AI AND ML LAB

- *Implement and demonstrate AI and ML algorithms.*
- *Evaluate different algorithms.*

LABORATORY OUTCOMES

CMR INSTITUTE OF TECHNOLOGY

Programs List

1. Implement A* Search algorithm.

2. Implement AO* Search algorithm.

3. For a given set of training data examples stored in a .CSV file, implement and demonstrate the Candidate-Elimination algorithm to output a description of the set of all hypotheses consistent with the training examples.

4. Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.

5. Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.

CMR INSTITUTE OF TECHNOLOGY

Programs List

6. Write a program to implement the naïve Bayesian classifier for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets.

7. Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using k-Means algorithm. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program.

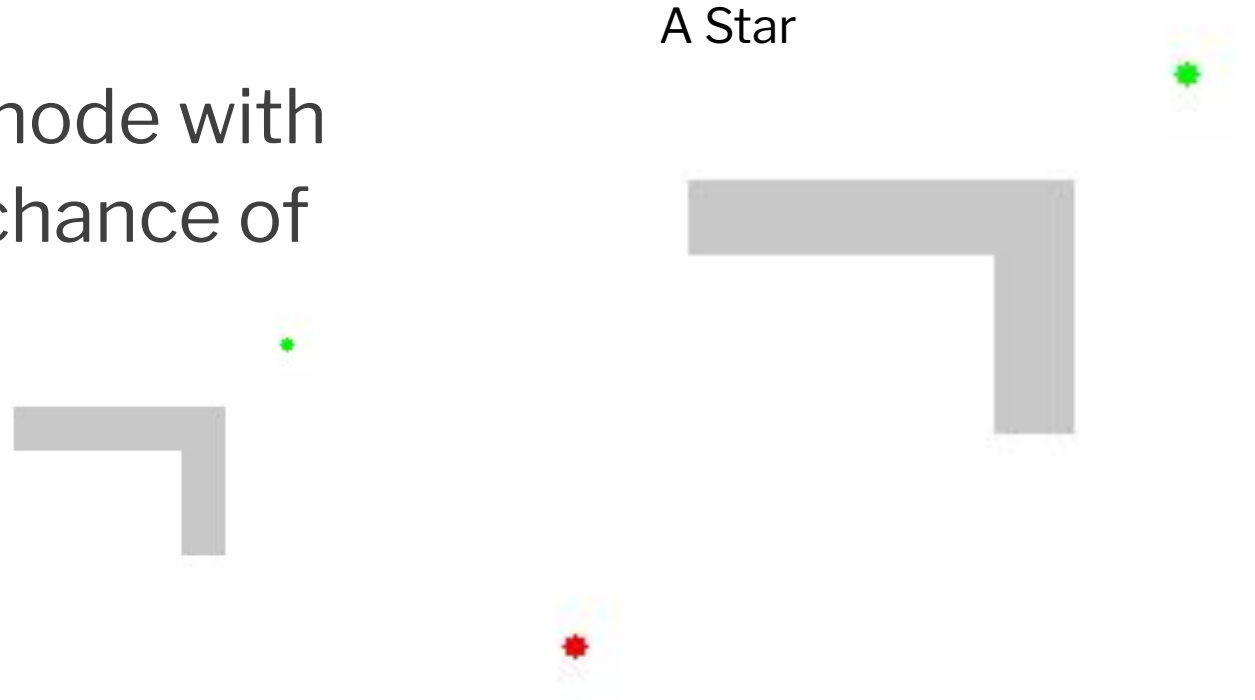
8. Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem.

9. Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs

A* Algorithm – originally proposed by Hart et al [1968;1972]

Best First search

Algorithm prioritizes the node with lowest f value and ‘best’ chance of reaching the goal node



A Star

Dijikstra’s

Image Source : Wikipedia

A* Algorithm – originally proposed by Hart et al [1968;1972]

Input : start node, goal node

OPEN list, CLOSED list

OPEN: nodes generated using heuristic function, but have not been examined yet. This is a priority queue in which element with the highest priority is used to manipulate the list.

CLOSED: nodes that have already been examined.

- important in a graph rather than a tree as we need to check if nodes have already been generated.

Functions

f = g+h the measure of cost from node to destination

g : measure of actual cost to go from initial state to goal state. It is not the estimate. It is the exact sum of costs applied along best path.

h: estimate of the additional cost from current node to goal node.

Function astar:

Input : start node, goal node, grid

Output : path

Create OPEN and CLOSED list

Set current = start

Append current to OPEN list

while the OPEN list is not empty {

Select node from OPEN list with minimum f where
f(current) = g(current) + h(current)

if current is goal we have found the solution;
while parent(current) exists:
Add parent to the path; Set current to parent
Invert the path from start to goal and return

Remove current from OPEN list and add it to CLOSED list

```
while the OPEN list is not empty {
    ......
    for each node in current's children {
            if  node in CLOSED list{
                Calculate new_cost  = g(current) + g(move_cost)
                 if new_cost<= g(node) { Move node from CLOSED list to OPEN list }

             } else if node in OPEN list {
                Calculate new_cost  = g(current) + g(move_cost)
                if new_cost <=g(node) {
                      Set g(node) to new_cost;  Set parent of node to current   }
            } else {
             Set g(node) = g(current) + g(move_cost)
                   Set h to heuristic distance from node to goal
             Set parent of node to current }
             Add child to OPEN list
        }
} // end while
```

Creating the grid and node object

- use 2 dimensional array.
- **use 0's and 1's,** 0: blocked 1: move is possible.
- **Create a class for Node.**
  - **__init__(self, value, coord)**
    - pass value(1 or 0) and coordinates(as a tuple) of the node and initialize them.
    - initialize g and h to 0, Initialize parent to None
    - Create a **move_cost(self,other)** function - We are working on a smooth terrain and all move costs are 1. But it may be modified with a cost matrix.
    - Write a **__str__ (self)**function to print the f,g,h and coord of the node.
- loop through all items in the grid and create nodes for each coordinate.

```
class Node:
    def __init__(self, value,coord):
        self.value = value
        self.coord = coord
        self.g=0; self.h=0 #initialization
        self.parent=None

    def __str__(self):
        s = f'{self.coord} f= {self.g+self.h:0.2f} \
        g={self.g:0.2f}, h= {self.h:0.2f}'
        return s

    def move_cost(self, other):
        return 1
```

```
#use case 1
grid = [[1,1,1,1], #1-not blocked, 0 -  not blocked
        [1,1,1,1],
        [1,1,1,1],
        [1,1,0,0],
        [1,1,0,1]]

#Convert all the points to instances of node
for x in range(len(grid)):
    for y in range(len(grid[x])):
        grid[x][y] = Node(grid[x][y],(x,y))

start = grid[4][0]
goal = grid [0][3]
```

```
#use case 2
grid =[[1,1,0,0,0,1,1],
       [1,1,1,1,0,1,1],
       [1,1,1,1,0,1,1],
       [1,1,1,1,0,1,1],
       [1,1,0,0,0,1,1],
       [1,1,1,1,1,1,1],
       [1,1,1,1,1,1,1] ]

#Convert all the points to instances of node
for x in range(len(grid)):
    for y in range(len(grid[x])):
        grid[x][y] = Node(grid[x][y],(x,y))

start = grid[3][1]
goal = grid [5][5]
```

```
#use case 3
import numpy as np
grid = np.ones((25,35), dtype=int)
grid = grid.tolist()

#Convert all the points to instances of node
for x in range(len(grid)):
    for y in range(len(grid[x])):
        grid[x][y] = Node(grid[x][y],(x,y))

#25 by 35 use case 3
start = grid[10][1]
goal= grid[24][34]
```

Set start and goal state and call A* function (Driver Code) (Run this cell at the last)

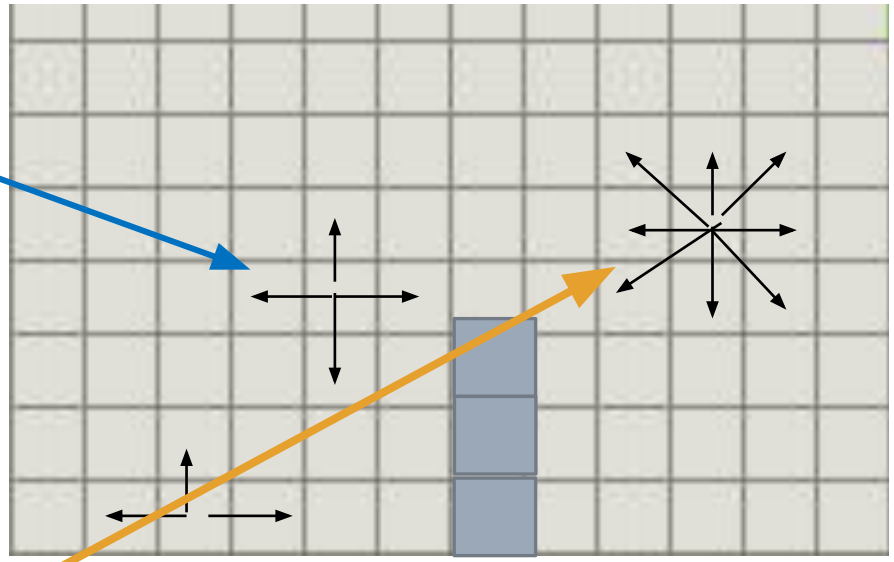
```
#Driver Code
path = aStar(start,goal ,grid)
if path:
    print("** Path ** ")
    for p in path:
        print(p.coord, end=" ")
else:
    print("No path found")
```

Functions used by A*

- generate_children
  - generates children based on moves allowed.
- heuristic – depending on moves allowed.

Generating children

- **4 moves (left, right, top bottom)**
- If it is blocked, then move is not possible.
- links = [(x-1, y),(x,y-1),(x,y+1),(x+1,y)]
  - Left (x-1, y)
  - Bottom(x, y-1)
  - Top(x, y+1)
  - Right(x+1, y)
- **8 Moves if Diagonal moves are permitted.**
- Diagonal Moves: (x+1,y+1),(x-1,y-1),(x+1,y-1),(x-1,y+1)



- 4 moves (left, right, top bottom)

```
def children(current_node,grid):
    x,y = current_node.coord
    links = [(x-1, y),(x,y-1),(x,y+1),(x+1,y)]
            #(x+1,y+1),(x-1,y-1),(x+1,y-1),(x-1,y+1)#diagonal moves, add in later.

    valid_links=[link for row in grid for link in row if link.value!=0]
    valid_children = [link for link in valid_links if link.coord in links]

    return valid_children
```

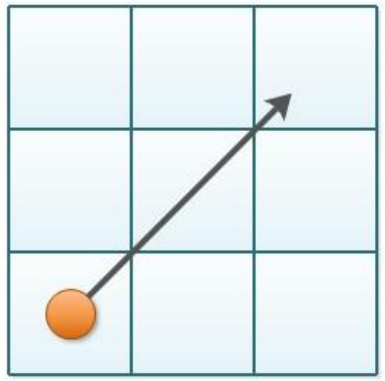
```
#For tracing purpose
    print('children of',current_node)
    for c in valid_children:
        print(c)
    ####
```

Which heuristic to use?

- On a square grid that allows **4 directions** of movement, use Manhattan distance.
- On a square grid that allows **8 directions** of movement, use Diagonal distance
- On a square grid that allows **any direction** of movement, you might or might not want Euclidean distance (L_2). If A* is finding paths on the grid but you are allowing movement not on the grid, you may want to consider other representations of the map.
- On a hexagon grid that allows **6 directions** of movement, use Manhattan distance adapted to hexagonal grids.

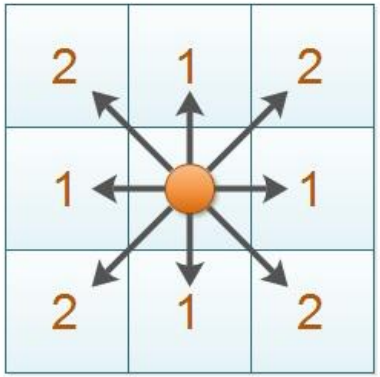
Which heuristic to use?

Euclidean Distance



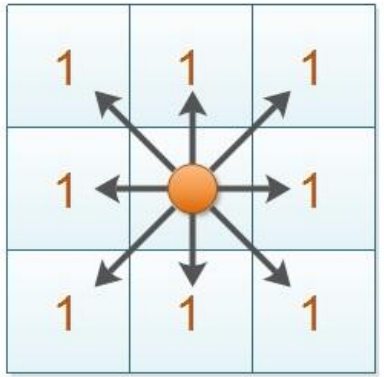
$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Manhattan Distance



$$|x_1 - x_2| + |y_1 - y_2|$$

Chebyshev Distance

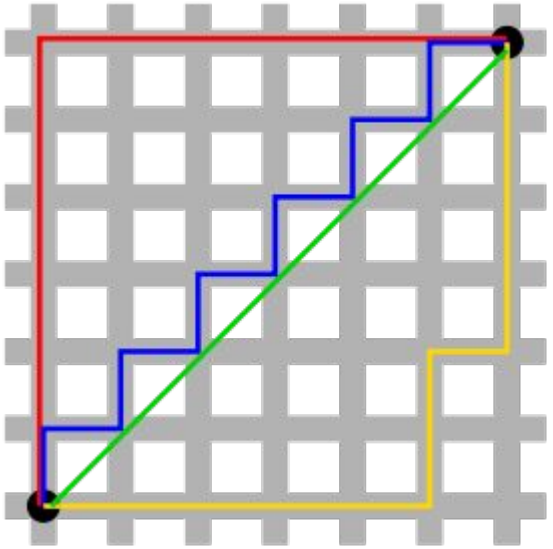


$$\max(|x_1 - x_2|, |y_1 - y_2|)$$

Source : https://iq.opengenus.org/euclidean-vs-manhattan-vs-chebyshev-distance/

Manhattan Heuristic – for 4 moves

```
# for 4 moves
def manhattan(node, goal):
    #manhattan distance
    xN,yN = node.coord
    xG,yG = goal.coord
    h = abs(xN-xG) + abs(yN-yG)
    return h
```



Diagonal Heuristic – for 8 moves

We compute the number of steps you take if you can't take a diagonal,

Then subtract the steps you save by using the diagonal.

There are min(dx, dy) diagonal steps, and each one costs D2 but saves you 2×D non-diagonal steps.

When D = 1 and D2 = 1, this is called the Chebyshev distance.

When D = 1 and D2 = sqrt(2), this is called the *octile distance*.

```
function heuristic(node) =
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
```

Diagonal Chebyshev Heuristic – for 8 moves

```
# for 8 moves
def diagonal(node, goal):
    xN,yN = node.coord
    xG,yG  = goal.coord
    dx = abs(xN - xG)
    dy = abs(yN- yG)
    return (dx + dy) - min(dx, dy)
```

```
def aStar(start, goal, grid):
    #The open and closed lists
    OPEN = list(); CLOSED=list()
    #Set current node to start node
    current = start
    #Add start node to the OPEN list
    OPEN.append(current)
    i=0 # for tracing purpose

    #While the open list is not empty
    while OPEN:
        print('Iteration ',i) # for tracing purpose
        i+=1 # for tracing purpose

        #Find the item in the open set with the lowest g + h score
        current = min(OPEN, key=lambda o:o.g + o.h)
        # print statements for tracing purpose
        print('Current Node', current)
```

```
#If it is the item we want, retrace the path and return it
if current == goal: # trace path by using parent link
    path = []
    while current.parent:
        path.append(current)
        current = current.parent
    path.append(current)
    return path[::-1]


#Move item from OPEN to CLOSED
OPEN.remove(current); CLOSED.append(current)
```

```
    #Loop through the node's children/siblings
    for node in children(current,grid):
        #If it is already in the closed list and updated cost is lower, move to OPEN list
        if node in CLOSED:
            new_cost = current.g + current.move_cost(node)
            if new_cost<=node.g:
                OPEN.append(node);CLOSED.remove(node)

        #Otherwise if it is already in the open set
        elif node in OPEN:
            #Check if we beat the G score
            new_cost = current.g + current.move_cost(node)
            if new_cost<=node.g:
                #If so, update the node to have a new parent
                node.g = new_cost
                node.parent = current
        else:
            #If it isn't in the open set, calculate the G and H score for the node
            node.g = current.g + current.move_cost(node)
            node.h = manhattan(node, goal)
            #Set the parent to our current item
            node.parent = current
            #Add it to the list
            OPEN.append(node)
#If no path found
return None
```

Breaking Ties

- either adjust the g or h values.
- needs to be deterministic with respect to the vertex (i.e., it shouldn't be a random number)
- it needs to make the f values differ
- If we scale it downwards, then f will increase as we move towards the goal.
- this means that A* will prefer to expand vertices close to the starting point instead of vertices close to the goal.
- We can instead scale h upwards slightly (even by 0.1%). A* will prefer to expand vertices close to the goal.
- Assuming that you don't expect the paths to be more than 1000 steps long, you can choose p = 1/1000

Heuristic used to control A*'s behavior

The heuristic can be used to control A*'s behavior.

At one extreme, if h(n) is 0, then only g(n) plays a role, and A* turns into Dijkstra's Algorithm, which is guaranteed to find a shortest path.

If h(n) is always lower than (or equal to) the cost of moving from n to the goal, then A* is guaranteed to find a shortest path. The lower h(n) is, the more node A* expands, making it slower.

If h(n) is exactly equal to the cost of moving from n to the goal, then A* will only follow the best path and never expand anything else, making it very fast. Although you can't make this happen in all cases, you can make it exact in some special cases. It's nice to know that given perfect information, A* will behave perfectly.

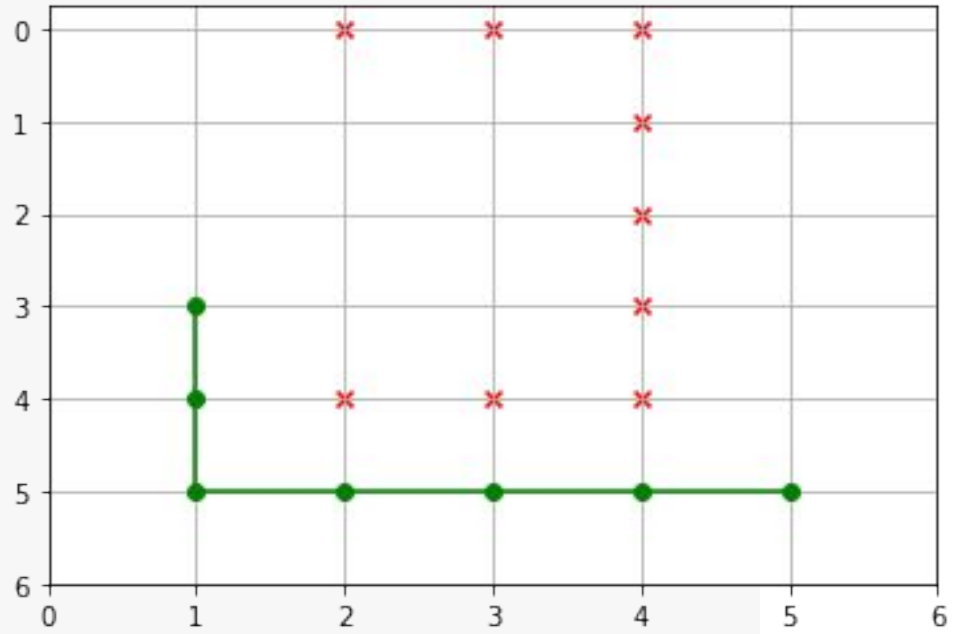
If h(n) is sometimes greater than the cost of moving from n to the goal, then A* is not guaranteed to find a shortest path, but it can run faster.At the other extreme, if h(n) is very high relative to g(n), then only h(n) plays a role, and A* turns into Greedy Best-First-Search.

Trace output

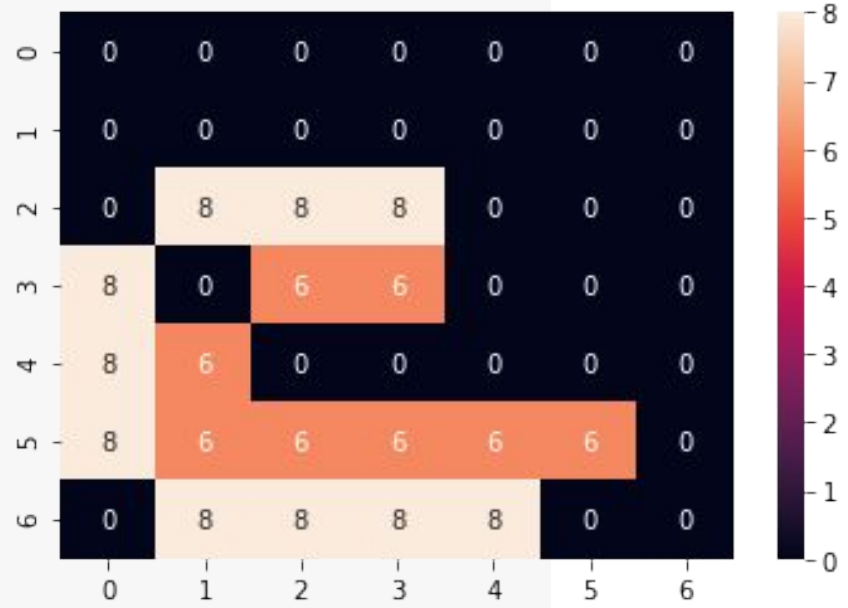
Draw grid, mention coordinates and f,g,h values of all the cells explored.

Use visualization code to help with tracing.

```
# Visualization  - for your understanding
# not required for exam.
#Obstacles
invalid_links=[link.coord[::-1] for row in grid for link in row if link.value==0]
import matplotlib.markers as mp
import matplotlib.pyplot as plt
import numpy as np

a=[]
plt.gca().invert_yaxis()

yint = range(0,len(grid))
xint = range(0, len(grid[0]))

if path:
    print("** Path ** ")
    for p in path:
        a.append(p.coord[::-1])
    plt.plot(*zip(*a), marker='o', color='green')
    #obstacles
    if invalid_links:
        plt.scatter(*zip(*invalid_links), marker='x', color='red')
    plt.yticks(yint);plt.xticks(xint)
    plt.grid(True)
    plt.show()
```



```
import seaborn as sns

f_vals = []
for row in grid:
    vals=[]
    for col in row:
        vals.append(col.g+col.h)
    f_vals.append(vals)


ax = sns.heatmap(f_vals, annot=True, fmt="d")
```



Further Reading

http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html