**­­**

|  |  |  |  |
| --- | --- | --- | --- |
| Sr.  No. | Aim | Page no. | Signature |
| 1 | Implementation of Simple Search 1 algorithm. | 1 |  |
| 2 | Implementation of Simple Search 2 algorithm. | 3 |  |
| 3 | Implementation of Simple Search 3 algorithm. | 5 |  |
| 4 | Implementation of Breadth First Search algorithm. | 8 |  |
| 5 | Implementation of Depth First Search algorithm. | 10 |  |
| 6 | Implementation of A \* algorithm. | 12 |  |
| 7 | Implementation of Best First Search algorithm. | 15 |  |
| 8 | Implementation of Support Vector Machine algorithm. | 18 |  |
| 9 | Implementation of Decision Tree. | 20 |  |
| 10 | Implementation of Adaboost ensemble learning algorithm. | 23 |  |

Aim:

## 1)Implementation of Simple Search 1 algorithm

Code:

import random

OPEN = ['S']

map\_list = {'S': ['A', 'B', 'C'],

'A': ['S', 'D'],

'B': ['S', 'E'],

'C': ['S', 'F'],

'D': ['A', 'G'],

'E': ['B', 'G', 'F'],

'F': ['C', 'E'],

'G': ['D', 'E']}

def moregen(node):

return map\_list[node]

def goaltest(node):

return node == 'G'

def ss1():

while len(OPEN) > 0:

random.shuffle(OPEN)

N = OPEN.pop()

if goaltest(N):

return "Found"

else:

n = moregen(N)

for i in n:

if i not in OPEN:

OPEN.append(i)

print("OPEN\_LIST", OPEN)

return "NOT FOUND"

print(ss1())

2)Aim:

## Implementation of Simple Search 2 algorithm

Code:

|  |
| --- |
| import random  OPEN = ['S']  CLOSED = []  map\_list = {'S': ['A', 'B', 'C'],  'A': ['S', 'D'],  'B': ['S', 'E'],  'C': ['S', 'F'],  'D': ['A', 'G'],  'E': ['B', 'G', 'F'],  'F': ['C', 'E'],  'G': ['D', 'E']}  def moregen(node):  return map\_list[node]  def goaltest(node):  return node == 'G'  def ss2():  while len(OPEN) > 0:  random.shuffle(OPEN)  N = OPEN.pop()  CLOSED.append(N)  if goaltest(N):  return "FOUND"  else:  n = moregen(N)  for i in n:  if i not in CLOSED and i not in OPEN:  OPEN.append(i)  print("OPEN\_LIST", OPEN)  print("CLOSED\_LIST", CLOSED)  return "NOT FOUND"  print(ss2()) |

3)Aim:

## Implementation of Simple Search 3 algorithm

Code:

import random

OPEN = [['S', None]]

CLOSED = []

map\_list = {'S': ['A', 'B', 'C'],

'A': ['S', 'D'],

'B': ['S', 'E'],

'C': ['S', 'F'],

'D': ['A', 'G'],

'E': ['B', 'G', 'F'],

'F': ['C', 'E'],

'G': ['D', 'E']}

def moregen(node):

return map\_list[node]

def goaltest(node):

return node == 'G'

def returnpath(path):

if path is not None:

return str(path[0]) + returnpath(path[1])

else:

return " "

def ss3():

while len(OPEN) > 0:

random.shuffle(OPEN)

print("OPEN\_LIST", OPEN)

M = OPEN.pop()

N = M[0]

CLOSED.append(N)

print("Picked: ", CLOSED)

if goaltest(N):

print("GOAL FOUND")

print("Path: ", returnpath(M)[::-1])

return

else:

neigh = moregen(N)

for node in neigh:

if node not in CLOSED and node not in OPEN:

new\_list = [node, M]

OPEN.append(new\_list)

print("NOT FOUND")

print(ss3())

4)Aim:

## Implementation of Breadth First Search algorithm

Code:

graph = {'S': ['A', 'B', 'C'],

'A': ['S', 'D'],

'B': ['S', 'E'],

'C': ['S', 'F'],

'D': ['A', 'G'],

'E': ['B', 'G', 'F'],

'F': ['C', 'E'],

'G': ['D', 'E']}

visited = []

queue = []

def bfs(visited, graph, node):

visited.append(node)

queue.append(node)

while queue:

m = queue.pop(0)

print(m, end=" ")

for neigh in graph[m]:

if neigh not in visited:

visited.append(neigh)

queue.append(neigh)

bfs(visited, graph, 'S')

5)Aim:

## Implementation of Depth First Search algorithm

Code:

graph = {'S': ['A', 'B', 'C'],

'A': ['S', 'D'],

'B': ['S', 'E'],

'C': ['S', 'F'],

'D': ['A', 'G'],

'E': ['B', 'G', 'F'],

'F': ['C', 'E'],

'G': ['D', 'E']}

visited = []

def dfs(visited, graph, node):

if node not in visited:

print(node)

visited.append(node)

for neigh in graph[node]:

dfs(visited, graph, neigh)

dfs(visited, graph, 'S')

6)Aim:

## Implementation of A \* algorithm

Code:

nodeList = {

'mumbai': [('delhi', 1200), ('nasik', 350), ('goa', 800), ('pune', 130)],

'delhi': [('nasik', 375), ('mumbai', 1200)],

'nasik': [('indore', 600), ('delhi', 375), ('mumbai', 350), ('nagpur', 600)],

'indore': [('nasik', 600)],

'nagpur': [('nasik', 600), ('pune', 450)],

'pune': [('mumbai', 130), ('nagpur', 450), ('blore', 550)],

'blore': [('hyd', 110), ('goa', 750)],

'goa': [('blore', 750), ('hyd', 850), ('mumbai', 800)],

'hyd': [('blore', 110), ('goa', 850)]

}

hd = {

'mumbai': 790, 'delhi': 1515, 'nasik': 1140, 'indore': 1540,

'nagpur': 1110, 'pune': 660, 'blore': 110, 'goa': 850, 'hyd': 0

}

openList = [('mumbai', 700)]

closedList = []

def moveGen(node):

return nodeList[node[0]]

def goalTest(node):

return node[0] == 'hyd'

def sort(mylist):

for i in range(len(mylist)):

for j in range(0, len(mylist) - i - 1):

if mylist[j][1] > mylist[j + 1][1]:

temp = mylist[j]

mylist[j] = mylist[j + 1]

mylist[j +1] = temp

return mylist

def AStar():

while len(openList) > 0:

sort(openList)

print("Open List Contains:", openList)

node = openList.pop(0)

closedList.append((node[0], hd[node[0]))

print("Picked node:", node)

if goalTest(node):

return "Goal Found"

else:

neighbours = moveGen(node)

print("Neighbours of", node[0], "are:", neighbours)

for node in neighbours:

if node not in openList and node[0] not in [i[0] for i in closedList]:

tup = (node[0], (node[1] + hd[node[0]))

openList.append(tup)

return "Goal Not Found"

result = AStar()

print(result)

7)Aim:

## Implementation of Best First Search algorithm

Code:

map\_list = {

'Mumbai': [('Pune', 750), ('Delhi', 1500), ('Goa', 1300)],

'Goa': [('Mumbai', 1200)],

'Delhi': [('Mumbai', 1200), ('Guwahati', 100), ('Pune', 750)],

'Chennai': [('Pune', 750)],

'Kolkata': [('Guwahati', 100), ('Pune', 750)],

'Pune': [('Mumbai', 1200), ('Kolkata', 0), ('Chennai', 1600), ('Delhi', 1500)],

'Guwahati': [('Delhi', 1500), ('Kolkata', 0)]

}

OPEN = [[('Mumbai', 1200), None]]

CLOSED = []

def movegen(node):

return map\_list[node]

def goaltest(node):

return node == 'Kolkata'

final = []

def reconstructpath(path):

if path is None:

return ""

else:

final.append(path[0][0])

reconstructpath(path[1])

return final

def sort(a):

for i in range(len(a)):

for j in range(0, len(a) - i - 1):

if a[j][0][1] > a[j + 1][0][1]:

a[j], a[j + 1] = a[j + 1], a[j]

return a

def best():

while len(OPEN) > 0:

print("Open List:", OPEN)

x = sort(OPEN)

seen = x.pop(0)

N = seen[0][0]

CLOSED.append(N)

print("Closed list contains", CLOSED)

print("Node Picked: ", N)

if goaltest(N):

print(reconstructpath(seen)[::-1])

return "Found"

else:

neigh = movegen(N)

for i in neigh:

if i[0] not in CLOSED and i not in OPEN:

new = [i, seen]

OPEN.append(new)

return "Not Found"

best()

8)Aim:

Implementation of Support Vector Machine algorithm.

Code:

import numpy as np

import pandas as pd

from sklearn import datasets

from sklearn.svm import SVC

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

balance\_data = pd.read\_csv('C:\\Users\\chira\\Desktop\\Code\\AI\\balancescale.data', sep=',', header=None)

print("Dataset Length:: ", len(balance\_data))

print("Dataset Shape:: ", balance\_data.shape)

print(balance\_data.head())

X = balance\_data.values[:, 1:5]

Y = balance\_data.values[:, 0]

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size=0.3, random\_state=100)

svclassifier = SVC(kernel='linear')

svclassifier.fit(X\_train, Y\_train)

print(Y\_test)

Y\_pred = svclassifier.predict(X\_test)

print(Y\_pred)

print(accuracy\_score(Y\_pred, Y\_test) \* 100)

9)Aim:

Implementation of Decision Tree.

Code:

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

from sklearn import tree

balance\_data = pd.read\_csv('C:\\Users\\chira\\Desktop\\Code\\AI\\balancescale.data', sep=',', header=None)

print("Dataset Length: ", len(balance\_data))

print("Dataset Shape: ", balance\_data.shape)

print(balance\_data.head())

X = balance\_data.values[:, 1:5]

Y = balance\_data.values[:, 0]

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size=0.4, random\_state=100)

clf\_entropy = DecisionTreeClassifier(criterion="entropy", random\_state=100, max\_depth=3, min\_samples\_leaf=5)

clf\_entropy.fit(X\_train, Y\_train)

clf\_gini = DecisionTreeClassifier(criterion="gini", random\_state=100, max\_depth=3, min\_samples\_leaf=5)

clf\_gini.fit(X\_train, Y\_train)

print(Y\_test)

y\_pred\_en = clf\_entropy.predict(X\_test)

y\_pred\_gini = clf\_gini.predict(X\_test)

print(y\_pred\_en)

print(y\_pred\_gini)

accuracy = accuracy\_score(y\_pred\_en, Y\_test) \* 100

print("Accuracy: ", accuracy)

10)Aim:

Implementation of Adaboost ensemble learning algorithm.

Code:

|  |
| --- |
| import pandas  from sklearn import model\_selection  from sklearn.ensemble import AdaBoostClassifier  url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv"  names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']  dataframe = pandas.read\_csv(url, names=names)  array = dataframe.values  X = array[:, 0:8]  Y = array[:, 8]  seed = 7  num\_trees = 30  kfold = model\_selection.KFold(n\_splits=10)  model = AdaBoostClassifier(n\_estimators=num\_trees, random\_state=seed)  results = model\_selection.cross\_val\_score(model, X, Y, cv=kfold)  print(results.mean() \* 100) |