CI Lab practise

BFS

In [85]:
```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph=defaultdict(list)
    def addEdge(self,u,v):
        self.graph[u].append(v)
    def BFS(self,s):
        visited=[False]*(max(self.graph)+1)
        queue=[]
        queue.append(s)
        visited[s]=True
        while queue:
            s=queue.pop(0)
            print(s,end="")
            for i in self.graph[s]:
                if visited[i]==False:
                    queue.append(i)
                    visited[i]=True

g=Graph()
g.addEdge(0,1)
g.addEdge(0,2)
g.addEdge(1,2)
g.addEdge(2,0)
g.addEdge(2,3)
g.addEdge(3,3)
print("BFS traversal from 2:")
g.BFS(2)
```

```
BFS traversal from 2:
2031
```

DFS

In [86]:
```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph=defaultdict(list)
    def addEdge(self,u,v):
        self.graph[u].append(v)
    def DFSUtil(self,v,visited):
        visited.add(v)
        print(v,end=' ')
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour,visited)
    def DFS(self,v):
        visited=set()
        self.DFSUtil(v,visited)

g=Graph()
g.addEdge(0,1)
g.addEdge(0,2)
g.addEdge(1,2)
g.addEdge(2,0)
g.addEdge(2,3)
g.addEdge(3,3)
print("DFS traversal from 2:")
g.DFS(2)
```

```
DFS traversal from 2:
2 0 1 3
```

Best First Search

In [88]:
```
from queue import PriorityQueue
v=14
graph=[[] for i in range(v)]
def best_first_search(source,target,n):
    visited=[0]*n
    visited[source]=True
    pq=PriorityQueue()
    pq.put((0,source))
    while pq.empty()==False:
        u=pq.get()[1]
        print(u,end=" ")
        if u==target:
            break
        for v,c in graph[u]:
            if visited[v]==False:
                visited[v]=True
                pq.put((c,v))
    print()
def addEdge(x,y,cost):
    graph[x].append((y,cost))
    graph[y].append((x,cost))

addEdge(0,1,3)
addEdge(0,2,6)
addEdge(0,3,5)
addEdge(1,4,9)
addEdge(1,5,8)
addEdge(2,6,12)
addEdge(2,7,14)
addEdge(3,8,7)
addEdge(8,9,5)
addEdge(8,10,6)
addEdge(9,11,1)
addEdge(9,12,10)
addEdge(9,13,2)
source=0
target=9
best_first_search(source,target,v)
```

```
0 1 3 2 8 9
```

A*

In [89]:
```
def Astar(start,Goal):
    open_list=set(start)
    closed_list=set()
    g={}
    parents={}
    g[start]=0
    parents[start]=start
    while len(open_list)>0:
        n=None
        for v in open_list:
            if n==None or g[v]+heuristic(v)<g[n]+heuristic(n):
                n=v
        if n==Goal or Graph[n]==None:
            pass
        else:
            for(m,weight) in get_neighbours(n):
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m]=n
                    g[m]=g[n]+weight
                else:
                    if g[m]>g[n]+weight:
                        g[m]=g[n]+weight
                        parents[m]=n
                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)
        if n==None:
            print('Path does not exist!')
            return None
        if n==Goal:
            path=[]
            while parents[n]!=n:
                path.append(n)
                n=parents[n]
            path.append(start)
            path.reverse()
            print('Path found:{}'.format(path))
            return path

        open_list.remove(n)
        closed_list.add(n)
    print('Path does not exist!')
    return None

def get_neighbours(v):
    if v in Graph:
        return Graph[v]
    else:
        return None

def heuristic(n):
    H_dist={'A':11,'B':6,'C':99,'D':1,'E':7,'G':0}
    return H_dist[n]

Graph={
```

```
    'A':[('B',2),('E',3)],
    'B':[('C',1),('G',9)],
    'C':None,
    'E':[('D',6)],
    'D':[('G',1)]
}
Astar('A','G')
```

```
Path found:['A', 'E', 'D', 'G']
```

Out[89]:
```
['A', 'E', 'D', 'G']
```

OR gate perceptron

In [30]:
```
import numpy as np
def unitStep(v):
    if v>=0:
        return 1
    else:
        return 0
def perceptronModel(x,w,b):
    v=np.dot(w,x)+b
    y=unitStep(v)
    return y
def OR_Logicfunction(x):
    w=np.array([1,1])
    b=-0.5
    return perceptronModel(x,w,b)

test1=np.array([0,1])
test2=np.array([1,1])
test3=np.array([0,0])
test4=np.array([1,0])

print("OR({},{})={}".format(0,1,OR_Logicfunction(test1)))
print("OR({},{})={}".format(1,1,OR_Logicfunction(test2)))
print("OR({},{})={}".format(0,0,OR_Logicfunction(test3)))
print("OR({},{})={}".format(1,0,OR_Logicfunction(test4)))
```

```
OR(0,1)=1
OR(1,1)=1
OR(0,0)=0
OR(1,0)=1
```

Hill climbing

In [43]:
```
import random
def randomSolution(tsp):
    cities=list(range(len(tsp)))
    solution=[]
    randomcity=cities[random.randint(0,len(cities)-1)]
    solution.append(randomcity)
    cities.remove(randomcity)
    return solution
def routeLength(tsp,solution):
    routeLength=0
    for i in range(len(solution)):
        routeLength+=tsp[solution[i-1]][solution[i]]
    return routeLength
def getNeighbours(solution):
    neighbours=[]
    for i in range(len(solution)):
        for j in range(i+1,len(solution)):
            neighbour=solution.copy()
            neighbour[i]=solution[j]
            neighbour[j]=solution[i]
            neighbours.append(neighbour)
    return neighbours
def getBestNeighbour(tsp,neighbours):
    bestrouteLength=routeLength(tsp,neighbours)
    bestNeighbour=neighbours
    for neighbour in neighbours:
        currentrouteLength=routeLengthLength(tsp,neighbour)
        if currentrouteLength<bestrouteLength:
            bestrouteLength=currentrouteLength
            bestNeighbour=neighbour
    return bestNeighbour,bestrouteLength
def hillclimbing(tsp):
    currentSolution=randomSolution(tsp)
    currentrouteLength=routeLength(tsp,currentSolution)
    neighbours=getNeighbours(currentSolution)
    bestNeighbour,bestNeighbourRouteLength=getBestNeighbour(tsp,neighbours)

    while bestNeighbourRouteLength<currentrouteLength:
        currentSolution=bestNeighbour
        currentrouteLength=bestNeighbourRouteLength
        neighbours=getNeighbours(currentSolution)
        bestNeighbour,bestNeighbourRouteLength=getBestNeighbour(tsp,neighbours)
    return currentSolution,currentrouteLength
def main():
    tsp=[
        [0,400,500,300],
        [400,0,300,500],
        [500,300,0,400],
        [300,500,400,0]
    ]
    print(hillclimbing(tsp))

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

```
([1], 0)
```

In [44]:
```
import time
targ = 1.5  # target of t
location = 1   #starting point
step = 0.9  # step change starting point
a=0
b=0
while abs(targ-location)>0.05:
    print(round(location,2),end=" ")
    if targ>location:
        if b==1: # If I already been at b
            b=0
            a=0
            step = step*(0.9)
            location =location +  abs(location*step)
            print ('\t increase ',end="")
        else:
            location =location +  abs(location*step)
            a=1
            print ('\t increase ',end="")
    else:
        if a==1:  #If I already been at a
            a=0
            b=0
            step = step*(0.9)
            location =location -  abs(location*step)
            print ('\t decrease ', end="")
        else:
            location = location - abs(location*step)
            b=1
            print ('\t decrease ',end="")
    #time.sleep(0.5)  # just so it will be easy to see the change..
    print ('\t ',round(location,2))
```

```
1        increase        1.9
1.9      decrease        0.36
0.36     increase        0.65
0.65     increase        1.18
1.18     increase        2.14
2.14     decrease        0.58
0.58     increase        1.0
1.0      increase        1.73
1.73     decrease        0.6
0.6      increase        0.99
0.99     increase        1.64
1.64     decrease        0.67
0.67     increase        1.07
1.07     increase        1.69
1.69     decrease        0.79
0.79     increase        1.22
1.22     increase        1.86
1.86     decrease        0.97
0.97     increase        1.44
1.44     increase        2.12
2.12     decrease        1.21
1.21     increase        1.73
```

```
1.73     decrease         1.06
1.06     increase         1.47
```

In [45]:
```
import time
target=1.5
step=0.9
a=0
b=0
location=1
while abs(target-location)>0.05:
    print(round(location,2),end="")
    if target>location:
        if b==1:
            a=0
            b=0
            step=step*(0.9)
            location=location+abs(location*step)
            print('\t increase ',end="")
        else:
            location=location+abs(location*step)
            a=1
            print('\t increase ',end="")
    else:
        if a==1:
            a=0
            b=0
            step=step*(0.9)
            location=location-abs(location*step)
            print('\t decrease ',end="")
        else:
            location=location-abs(location*step)
            b=1
            print('\t decrease ',end="")
    print('\t',round(location,2))
```

```
1        increase        1.9
1.9      decrease        0.36
0.36     increase        0.65
0.65     increase        1.18
1.18     increase        2.14
2.14     decrease        0.58
0.58     increase        1.0
1.0      increase        1.73
1.73     decrease        0.6
0.6      increase        0.99
0.99     increase        1.64
1.64     decrease        0.67
0.67     increase        1.07
1.07     increase        1.69
1.69     decrease        0.79
0.79     increase        1.22
1.22     increase        1.86
1.86     decrease        0.97
0.97     increase        1.44
1.44     increase        2.12
2.12     decrease        1.21
1.21     increase        1.73
1.73     decrease        1.06
1.06     increase        1.47
```

Genetic algo

In [ ]:

Linear Regression

In [104]:
```
import numpy as np
```

In [105]:
```
from sklearn.datasets import make_classification
X,y=make_classification(n_samples=100,n_classes=3,n_features=3,n_informative=3,n
```

In [106]:
```
from sklearn.model_selection import train_test_split
```

In [107]:
```
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.25,random_state=0
```

In [108]:
```
X_train.shape
```

Out[108]: (75, 3)

In [109]:
```
y_train.shape
```

Out[109]: (75,)

In [110]:
```
X_test.shape
```

Out[110]: (25, 3)

In [111]:
```
y_test.shape
```

Out[111]: (25,)

In [112]:
```
from sklearn.preprocessing import StandardScaler
sc_X=StandardScaler()
X_train=sc_X.fit_transform(X_train)
X_test=sc_X.transform(X_test)
```

In [113]:
```
from sklearn.linear_model import LogisticRegression
lr=LogisticRegression()
lr.fit(X_train,y_train)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:43
2: FutureWarning: Default solver will be changed to 'lbfgs' in 0.22. Specify a
solver to silence this warning.
  FutureWarning)
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:46
9: FutureWarning: Default multi_class will be changed to 'auto' in 0.22. Specif
y the multi_class option to silence this warning.
  "this warning.", FutureWarning)
```

Out[113]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

In [114]:
```
y_pred=lr.predict(X_test)
```

In [115]:
```
y_test
```

Out[115]:
```
array([2, 2, 1, 1, 2, 2, 0, 1, 0, 2, 1, 2, 1, 1, 0, 0, 2, 2, 2, 2, 1, 0,
       1, 2, 1])
```

In [116]:
```
y_pred
```

Out[116]:
```
array([2, 2, 1, 0, 2, 2, 0, 1, 0, 2, 1, 2, 0, 1, 0, 0, 2, 2, 2, 2, 2, 0,
       1, 2, 1])
```

In [117]:
```
from sklearn import metrics
accuracy=metrics.accuracy_score(y_test,y_pred)
print("Accuracy: ",accuracy)
print("Accuracy %: ",accuracy*100)
print("Rounding aacuracy : ",round(accuracy,2)*100)
```

```
Accuracy:  0.88
Accuracy %:  88.0
Rounding aacuracy :  88.0
```

Logistic Regression

In [ ]:
```
from sklearn.datasets import make_
```