MINMAX

```python
import math
def minimax(currentdepth,nodeindex,scores,maxturn,targetdepth):
    if(currentdepth==targetdepth):
        return scores[nodeindex]
    elif(maxturn):
        return max(minimax(currentdepth+1,nodeindex*2,scores,False,targetdepth),minimax(currentdepth+1,nodeindex*2+1,scores,False,targetdepth))
    else:
        return min(minimax(currentdepth+1,nodeindex*2,scores,True,targetdepth),minimax(currentdepth+1,nodeindex*2+1,scores,True,targetdepth))
scores=[3, 5, 2, 9, 12, 5, 23, 23]
td=math.log(len(scores),2)
print(minimax(0,0,scores,True,td))
```

```python
import math
def minimax(currentdepth,nodeindex,scores,maxturn,targetdepth):
   if(currentdepth==targetdepth):
      return scores[nodeindex]
   elif(maxturn):
      return max(minimax(currentdepth+1,nodeindex*2,scores,False,targetdepth),minimax(currentdepth+1,nodeindex*2+1,scores,False,targetdepth))
   else:
      return min(minimax(currentdepth+1,nodeindex*2,scores,True,targetdepth),minimax(currentdepth+1,nodeindex*2+1,scores,True,targetdepth))
scores=[3, 5, 2, 9, 12, 5, 23, 23]
td=math.log(len(scores),2)
print(minimax(0,0,scores,True,td))
```

ALPHA_BETA

```python
min1=-1000
max1=1000
def minimax(depth,nodeindex,values,maxturn,alpha,beta):
    if(depth==3):
        return values[nodeindex]
    if(maxturn):
        best=min1
        for i in range(2):
            value=minimax(depth+1,nodeindex*2+i,values,False,alpha,beta)
            best=max(best,value)
            alpha=max(best,alpha)
            if(alpha>=beta):
                break

        return best
    else:
        best=max1
        for i in range(2):
            value=minimax(depth+1,nodeindex*2+i,values,True,alpha,beta)
            best=min(best,value)
            beta=min(best,beta)
            if(alpha>=beta):
                break
        return best
values = [3, 5, 6, 9, 1, 2, 0, -1]
print("The optimal value is :", minimax(0, 0, values, True, min1,max1))
```

```python
min1=-1000
max1=1000
def minimax(depth,nodeindex,values,maxturn,alpha,beta):
   if(depth==3):
      return values[nodeindex]
   if(maxturn):
      best=min1
      for i in range(2):
         value=minimax(depth+1,nodeindex*2+i,values,False,alpha,beta)
         best=max(best,value)
         alpha=max(best,alpha)
         if(alpha>=beta):
            break
      return best
   else:
      best=max1
      for i in range(2):
         value=minimax(depth+1,nodeindex*2+i,values,True,alpha,beta)
         best=min(best,value)
```

```
            beta=min(best,beta)
            if(alpha>=beta):
                break
        return best
values = [3, 5, 6, 9, 1, 2, 0, -1]
print("The optimal value is :", minimax(0, 0, values, True, min1,max1))
```

HILL ClIMBING

```
import random

def randomSolution(tsp):
    cities = list(range(len(tsp)))
    solution = []

    for i in range(len(tsp)):
        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[0])
    bestNeighbour = neighbours[0]
    for neighbour in neighbours:
        currentRouteLength = routeLength(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()
```

BI DIRECTIONAL

```
class AdjacentNode:
    def __init__(self,node):
        self.vertex=node
        self.next=None
class BidirectionalSearch:
    def __init__(self,vertices):
        self.vertices=vertices
        self.graph=[None]*self.vertices
        self.src_queue=list()
        self.dest_queue=list()
        self.src_visited=[False]*self.vertices
        self.dest_visited=[False]*self.vertices
        self.src_parent=[None]*self.vertices

        self.dest_parent=[None]*self.vertices

    def add_edge(self,src,dest):
        node=AdjacentNode(dest)
        node.next=self.graph[src]
        self.graph[src]=node

        node=AdjacentNode(src)
        node.next=self.graph[dest]
        self.graph[dest]=node

    def bfs(self,direction):
        if(direction=='forward'):
            current=self.src_queue.pop(0)
            connected_node=self.graph[current]
            while connected_node:
                vertex=connected_node.vertex
```

```
            if not self.src_visited[vertex]:
                self.src_visited[vertex]=True
                self.src_queue.append(vertex)
                self.src_parent[vertex]=current
            connected_node=connected_node.next
    else:
        current=self.dest_queue.pop(0)
        connected_node=self.graph[current]
        while connected_node:
            vertex=connected_node.vertex
            if not self.dest_visited[vertex]:
                self.dest_visited[vertex]=True
                self.dest_queue.append(vertex)
                self.dest_parent[vertex]=current
            connected_node=connected_node.next
def is_intersecting(self):
    for i in range(self.vertices):
        if(self.src_visited[i] and self.dest_visited[i]):
            return i
    return -1


def print_path(self, intersecting_node,src, dest):
    path = list()
    path.append(intersecting_node)
    i = intersecting_node
    while i != src:
        path.append(self.src_parent[i])
        i = self.src_parent[i]
    path = path[::-1]
    i = intersecting_node
    while i != dest:
        path.append(self.dest_parent[i])
        i = self.dest_parent[i]
    print("*****Path*****")
    path = list(map(str, path))
    print(' '.join(path))

def bis(self,src,dest):
    self.src_queue.append(src)
    self.src_visited[src]=True
    self.src_parent[src]=-1

    self.dest_queue.append(dest)
    self.dest_visited[dest]=True
    self.dest_parent[dest]=-1

    while self.src_queue and self.dest_queue:
        self.bfs('forward')
        self.bfs('backward')
        intersecting_node = self.is_intersecting()
        if intersecting_node != -1:
            print(f"Path exists between {src} and {dest}")
            print(f"Intersection at : {intersecting_node}")
            self.print_path(intersecting_node,src, dest)
            exit(0)
    return -1
```

```
n = 15
src = 0
dest = 14
graph = BidirectionalSearch(n)
graph.add_edge(0, 4)
graph.add_edge(1, 4)
graph.add_edge(2, 5)
graph.add_edge(3, 5)
graph.add_edge(4, 6)
graph.add_edge(5, 6)
graph.add_edge(6, 7)
graph.add_edge(7, 8)
graph.add_edge(8, 9)
graph.add_edge(8, 10)
graph.add_edge(9, 11)
graph.add_edge(9, 12)
graph.add_edge(10, 13)
graph.add_edge(10, 14)

out = graph.bis(src, dest)

if out == -1:
        print(f"Path does not exist between {src} and {dest}")
```

MONTY HALL

```
import math
from pomegranate import *

# Initially the door selected by the guest is completely random
guest =DiscreteDistribution( { 'A': 1./3, 'B': 1./3, 'C': 1./3 } )

# The door containing the prize is also a random process
prize =DiscreteDistribution( { 'A': 1./3, 'B': 1./3, 'C': 1./3 } )

# The door Monty picks, depends on the choice of the guest and the prize door
monty =ConditionalProbabilityTable(
[[ 'A', 'A', 'A', 0.0 ],
[ 'A', 'A', 'B', 0.5 ],
[ 'A', 'A', 'C', 0.5 ],
[ 'A', 'B', 'A', 0.0 ],
[ 'A', 'B', 'B', 0.0 ],
[ 'A', 'B', 'C', 1.0 ],
[ 'A', 'C', 'A', 0.0 ],
[ 'A', 'C', 'B', 1.0 ],
[ 'A', 'C', 'C', 0.0 ],
[ 'B', 'A', 'A', 0.0 ],
[ 'B', 'A', 'B', 0.0 ],
[ 'B', 'A', 'C', 1.0 ],
```

```
[ 'B', 'B', 'A', 0.5 ],
[ 'B', 'B', 'B', 0.0 ],
[ 'B', 'B', 'C', 0.5 ],
[ 'B', 'C', 'A', 1.0 ],
[ 'B', 'C', 'B', 0.0 ],
[ 'B', 'C', 'C', 0.0 ],
[ 'C', 'A', 'A', 0.0 ],
[ 'C', 'A', 'B', 1.0 ],
[ 'C', 'A', 'C', 0.0 ],
[ 'C', 'B', 'A', 1.0 ],
[ 'C', 'B', 'B', 0.0 ],
[ 'C', 'B', 'C', 0.0 ],
[ 'C', 'C', 'A', 0.5 ],
[ 'C', 'C', 'B', 0.5 ],
[ 'C', 'C', 'C', 0.0 ]], [guest, prize] )

d1 = State( guest, name="guest" )
d2 = State( prize, name="prize" )
d3 = State( monty, name="monty" )

#Building the Bayesian Network
network = BayesianNetwork( "Solving the Monty Hall Problem With Bayesian Networks" )
network.add_states(d1, d2, d3)
network.add_edge(d1, d3)
network.add_edge(d2, d3)
network.bake()

beliefs = network.predict_proba({ 'guest' : 'B' })
#beliefs = map(str, beliefs)
print("".join( "{}{}".format( state.name, belief ) for state, belief in zip( network.states, beliefs ) ))
```

Q LEARNING

```
import numpy as np
import pylab as pl
import networkx as nx
edges = [(0, 1),(1, 5),(5, 6),(5, 4),(1, 2),
        (1, 3),(9, 10),(2, 4),(0, 6),(6, 7),
        (8, 9),(7, 8),(1, 7),(3, 9)]
goal=10
G=nx.Graph()
G.add_edges_from(edges)
pos=nx.spring_layout(G)
nx.draw_networkx_edges(G,pos)
nx.draw_networkx_labels(G,pos)
nx.draw_networkx_nodes(G,pos)
```

```
pl.show()
MATRIX_SIZE = 11
M = np.matrix(np.ones(shape =(MATRIX_SIZE, MATRIX_SIZE)))
M *= -1
for point in edges:
   if point[1] == goal:
      M[point] = 100
   else:
      M[point] = 0
   if point[0] == goal:
      M[point[::-1]] = 100
   else:
      M[point[::-1]]= 0
M[goal, goal]= 100
print(M)
Q = np.matrix(np.zeros([MATRIX_SIZE, MATRIX_SIZE]))

gamma = 0.75
initial_state = 1

def available_actions(state):
   current_state_row = M[state, ]
   available_action = np.where(current_state_row >= 0)[1]
   return available_action

available_action = available_actions(initial_state)

def sample_next_action(available_actions_range):
   next_action = int(np.random.choice(available_action, 1))
   return next_action


action = sample_next_action(available_action)

def update(current_state, action, gamma):

  max_index = np.where(Q[action, ] == np.max(Q[action, ]))[1]

  if max_index.shape[0] > 1:
     max_index = int(np.random.choice(max_index, size = 1))
  else:
     max_index = int(max_index)

  max_value = Q[action, max_index]
  Q[current_state, action] = M[current_state, action] + gamma * max_value
  if (np.max(Q) > 0):
   return(np.sum(Q / np.max(Q)*100))
  else:
   return (0)

update(initial_state, action, gamma)
```

```
scores = []
for i in range(1000):
    current_state = np.random.randint(0, int(Q.shape[0]))
    available_action = available_actions(current_state)
    action = sample_next_action(available_action)
    score = update(current_state, action, gamma)
    scores.append(score)

current_state = 0
steps = [current_state]

while current_state != 10:
    next_step_index = np.where(Q[current_state, ] == np.max(Q[current_state, ]))[1]
    if next_step_index.shape[0] > 1:
        next_step_index = int(np.random.choice(next_step_index, size = 1))
    else:
        next_step_index = int(next_step_index)
    steps.append(next_step_index)
    current_state = next_step_index

print("Most efficient path:")
print(steps)

pl.plot(scores)
pl.xlabel('No of iterations')
pl.ylabel('Reward gained')
pl.show()
```

BURGLARY

```
import pandas as pd
import numpy as np

from pgmpy.models import BayesianNetwork
from pgmpy.inference import VariableElimination

alarm_model = BayesianNetwork(
    [
        ("Burglary", "Alarm"),
        ("Earthquake", "Alarm"),
        ("Alarm", "JohnCalls"),
        ("Alarm", "MaryCalls"),
    ]
)

# Defining the parameters using CPT
from pgmpy.factors.discrete import TabularCPD

cpd_burglary = TabularCPD(
    variable="Burglary", variable_card=2, values=[[0.999], [0.001]]
```

```
)
cpd_earthquake = TabularCPD(
    variable="Earthquake", variable_card=2, values=[[0.998], [0.002]]
)
cpd_alarm = TabularCPD(
    variable="Alarm",
    variable_card=2,
    values=[[0.999, 0.71, 0.06, 0.05], [0.001, 0.29, 0.94, 0.95]],
    evidence=["Burglary", "Earthquake"],
    evidence_card=[2, 2],
)
cpd_johncalls = TabularCPD(
    variable="JohnCalls",
    variable_card=2,
    values=[[0.95, 0.1], [0.05, 0.9]],
    evidence=["Alarm"],
    evidence_card=[2],
)
cpd_marycalls = TabularCPD(
    variable="MaryCalls",
    variable_card=2,
    values=[[0.1, 0.7], [0.9, 0.3]],
    evidence=["Alarm"],
    evidence_card=[2],
)

# Associating the parameters with the model structure
alarm_model.add_cpds(
    cpd_burglary, cpd_earthquake, cpd_alarm, cpd_johncalls, cpd_marycalls
)
values = pd.DataFrame(np.random.randint(low=0, high=2, size=(100, 5)),
                      columns=["Burglary", "Alarm","Earthquake","JohnCalls", "MaryCalls"])
predict_data = values[80:]
predict_data = predict_data.copy()
predict_data.drop("Earthquake", axis=1, inplace=True)

y_prob = alarm_model.predict_probability(predict_data)

print(y_prob)
```