Exp – 1 Informed Search

A*

Code

```
from collections import deque

class Graph:

   def __init__(self, adjacency_list):
       self.adjacency_list = adjacency_list

   def get_neighbors(self, v):
       return self.adjacency_list[v]

   # heuristic function with equal values for all nodes
   def h(self, n):
       H = {
           'A': 1,
           'B': 1,
           'C': 1,
           'D': 1
       }

       return H[n]

   def a_star_algorithm(self, start_node, stop_node):
       # open_list is a list of nodes which have been visited, but who's 
neighbors
       # haven't all been inspected, starts off with the start node
       # closed_list is a list of nodes which have been visited
       # and who's neighbors have been inspected
       open_list = set([start_node])
       closed_list = set([])

       # g contains current distances from start_node to all other nodes
       # the default value (if it's not found in the map) is +infinity
       g = {}

       g[start_node] = 0

       # parents contains an adjacency map of all nodes
       parents = {}
```

```python
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None

            # find a node with the lowest value of f() - evaluation function
            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                    n = v;

            if n == None:
                print('Path does not exist!')
                return None

            # if the current node is the stop_node
            # then we begin reconstructin the path from it to the start_node
            if n == stop_node:
                reconst_path = []

                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]

                reconst_path.append(start_node)

                reconst_path.reverse()

                print('Path found: {}'.format(reconst_path))
                return reconst_path

            # for all neighbors of the current node do
            for (m, weight) in self.get_neighbors(n):
                # if the current node isn't in both open_list and closed_list
                # add it to open_list and note n as it's parent
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight

                # otherwise, check if it's quicker to first visit n, then m
                # and if it is, update parent data and g data
                # and if the node was in the closed_list, move it to open_list
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
```

```python
                        parents[m] = n

                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)

            # remove n from the open_list, and add it to closed_list
            # because all of his neighbors were inspected
            open_list.remove(n)
            closed_list.add(n)

        print('Path does not exist!')
        return None
adjacency_list = {
   'A': [('B', 1), ('C', 3), ('D', 7)],
   'B': [('D', 5)],
   'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')
```

Output

```
PS D:\exp> python -u "d:\exp\exp3_uninformed.py"
Path found: ['A', 'B', 'D']
PS D:\exp>
```

Exp 2 - Uninformed Search

DLS

Code

```python
graph={
    'S':['A','B'],
    'A':['C','D'],
    'B':['I','J'],
    'C':['E','F'],
```

```python
    'D':['G'],
    'I':['H'],
    'J':[]
}
def dls(start,goal,path,level,maxLimit):
    print('\nCurrent level -->',level)
    print('Goal node testing',start)
    path.append(start)
    if start==goal:
        print('Test successfull goal found')
        return path
    print('Goal node test failed')
    if level==maxLimit:
        return False
    print('Expanding current node:',start)
    for child in graph[start]:
        if dls(child, goal, path, level+1, maxLimit):
            return path
    return False

start='S'
goal=input('Enter goal:')
maxLimit=int(input("Enter max limit:"))
print()
path=list()
res=dls(start, goal, path, 0, maxLimit)
if(res):
    print('Path exists')
    print('Path',path)
else:
    print('Path doesnt exist')
```

Output

```
PS D:\exp> python -u "d:\exp\exp3_uninformed.py"
Enter goal:D
Enter max limit:3


Current level --> 0
Goal node testing S
Goal node test failed
Expanding current node: S

Current level --> 1
Goal node testing A
Goal node test failed
Expanding current node: A

Current level --> 2
Goal node testing C
Goal node test failed
Expanding current node: C

Current level --> 3
Goal node testing E
Goal node test failed

Current level --> 3
Goal node testing F
Goal node test failed
```

```
Current level --> 2
Goal node testing D
Test successfull goal found
Path exists
Path ['S', 'A', 'C', 'E', 'F', 'D']
PS D:\exp>
```

Exp – 3 Minmax Algorithm

Code

```
import math
def fun_minmax(cd, node, maxt, scr, td):
    if(cd == td):
        return scr[node]
    if(maxt):
        return max(fun_minmax(cd+1, node*2, False, scr, td),fun_minmax(cd+1,
node*2+1, False, scr, td))
    else:
        return min(fun_minmax(cd+1, node*2, True, scr, td),fun_minmax(cd+1,
node*2+1, True, scr, td))


scr = []
x =int(input("Enter total number of leaf Node = "))
for i in range(x):
    y = int(input("Enter leaf value: "))
    scr.append(y)

td = math.log(len(scr), 2)
```

```
cd = int(input("Enter current depth value: "))
nodev = int(input("Enter node value: "))
maxt = True

print("The answer is: ", end=" ")
answer = fun_minmax(cd, nodev, maxt, scr, td)
print(answer)
```

Output

```
PS D:\exp> python -u "d:\exp\exp3_uninformed.py"
Enter total number of leaf Node = 8
Enter leaf value: 10
Enter leaf value: 20
Enter leaf value: 30
Enter leaf value: 40
Enter leaf value: 50
Enter leaf value: 60
Enter leaf value: 70
Enter leaf value: 80
Enter current depth value: 0
Enter node value: 0
The answer is:  60
PS D:\exp>
```

Exp – 4 Alpha Beta Pruning

Code

```
# Initial values of Alpha and Beta
MAX, MIN = 1000, -1000
# Returns optimal value for current player
#(Initially called for root and maximizer)
def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
    # Terminating condition. i.e
    # leaf node is reached
    if depth == 3:
        return values[nodeIndex]
    if maximizingPlayer:
        best = MIN
        # Recur for left and right children
        for i in range(0, 2):
```

```
            val = minimax(depth + 1, nodeIndex * 2 + i,False, values, alpha,
beta)
            best = max(best, val)
            alpha = max(alpha, best)
            # Alpha Beta Pruning
            if beta <= alpha:
                break
        return best
    else:
        best = MAX
        # Recur for left and
        # right children
        for i in range(0, 2):
            val = minimax(depth + 1, nodeIndex * 2 + i,True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)
            # Alpha Beta Pruning
            if beta <= alpha:
                break
        return best

if __name__ == "__main__":

    values = [4, 2, 6, 19, 1, -2, 3, -1]
    print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

Output

```
PS D:\exp> python -u "d:\exp\exp3_uninformed.py"
The optimal value is : 4
PS D:\exp>
```

Exp 5 - Constraint Satisfaction Problem

graph coloring

Code:

```
colors = ['Red','Blue','Green']
states = ['Nagpur','Thane','Pune','Mumbai']
neighbors = {}
```

```python
neighbors['Nagpur'] = ['Thane','Pune']
neighbors['Thane'] = ['Nagpur','Pune','Mumbai']
neighbors['Pune'] = ['Nagpur','Thane','Mumbai']
neighbors['Mumbai'] = ['Thane','Pune']

colors_of_states = {}

def promising(state, color):
    for neighbor in neighbors.get(state):
        color_of_neighbor = colors_of_states.get(neighbor)
        if color_of_neighbor == color:
            return False
    return True

def get_color_for_state(state):
    for color in colors:
        if promising(state, color):
            return color

def main():
    for state in states:
        colors_of_states[state] = get_color_for_state(state)
    print(colors_of_states)

main()
```

Output

```
PS D:\exp> python -u "d:\exp\exp3_uninformed.py"
{'Nagpur': 'Red', 'Thane': 'Blue', 'Pune': 'Green', 'Mumbai': 'Red'}
PS D:\exp>
```

N Queens

Code:

```python
# Taking number of queens as input from user
N = int(input("Enter the number of queens: "))

# here we create a chessboard
# NxN matrix with all elements set to 0
```

```python
board = [[0]*N for _ in range(N)]

def attack(i, j):
    #checking vertically and horizontally if there are any queen placed
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonally if there are any queen placed
    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False

def N_queens(n):
    if n==0:
        return True
    # here we are checking whether we can place queen at ith row and jth column
    for i in range(0,N):
        for j in range(0,N):
            if (not(attack(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
                if N_queens(n-1)==True:
                    return True
                board[i][j] = 0

    return False

N_queens(N)
for i in board:
    print (i)
```

Output

```
PS D:\exp> python -u "d:\exp\exp3_uninformed.py"
Enter the number of queens: 8
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
PS D:\exp>
```

Sudoku

Code

```
N = 9
def printing(arr):
    for i in range(N):
        for j in range(N):
            print(arr[i][j], end=" ")
        print()

# Checks whether it will be legal to assign num to the given row, col


def isSafe(grid, row, col, num):

    # Check if we find the same num in the similar row, we return false
    for x in range(9):
        if grid[row][x] == num:
            return False

    # Check if we find the same num in the similar colum, we return false
    for x in range(9):
        if grid[x][col] == num:
            return False

    # Check if we find the same num in the particular 3*3 matrix , we return
false
    startRow = row-row % 3
    startCol = col-col % 3
    for i in range(3):
        for j in range(3):
```

```
            if grid[i+startRow][j + startCol] == num:
                return False
    return True

# Takes a partially filled-in grid and attempts to assign value to all unassigned
locations in such a way to meet the requirements for

def solveSudoku(grid, row, col):
    # Check if we have 8th row and 9th column(0 indexed matrix)
    if (row == N-1 and col == N):
        return True

    if col == N:
        row += 1
        col = 0

    # Check if the current position of the grid already contains value > 0 we
iterate for next column
    if grid[row][col] > 0:
        return solveSudoku(grid, row, col+1)

    for num in range(1, N+1, 1):
        if isSafe(grid, row, col, num):
            grid[row][col] = num
            if solveSudoku(grid, row, col + 1):
                return True
        grid[row][col] = 0
    return False


grid = [[3, 0, 6, 5, 0, 8, 4, 0, 0],
        [5, 2, 0, 0, 0, 0, 0, 0, 0],
        [0, 8, 7, 0, 0, 0, 0, 3, 1],
        [0, 0, 3, 0, 1, 0, 0, 8, 0],
        [9, 0, 0, 8, 6, 3, 0, 0, 5],
        [0, 5, 0, 0, 9, 0, 6, 0, 0],
        [1, 3, 0, 0, 0, 0, 2, 5, 0],
        [0, 0, 0, 0, 0, 0, 0, 7, 4],
        [0, 0, 5, 2, 0, 6, 3, 0, 0]]

if (solveSudoku(grid, 0, 0)):
    printing(grid)
else:
    print("no solution exists")
```

Output

```
PS D:\exp> python -u "d:\exp\exp3_uninformed.py"
3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
2 6 3 4 1 5 9 8 7
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9
PS D:\exp>
```

Exp - 6 Local Search

Code

```
from random import *
import random
import numpy
import copy

countCities = 20;
# 2D Array
cities = numpy.zeros(shape=(20,20))
# tour
hypothesis = [int]*countCities
visitedCities = []
saveState = []

threshold = 2
lastFitness = 0
trials = 0
cityIndex = 1

# calculates fitness based on the difference between the distances
def getFitness(fitness, hypothesis, saveState, cities):
    oldDistance = getDistance(cities, saveState)
    print("Old Distance ",oldDistance,"km")
    print("")
    newDistance = getDistance(cities, hypothesis)
    print("New Distance ",newDistance,"km")
    print("")
    if(oldDistance > newDistance):
```

```python
        fitness += 1
    elif(oldDistance < newDistance):
        fitness -= 1

    return fitness

# choose random City at position cityIndex
def doRandomStep():
    global visitedCities
    global saveState
    global hypothesis
    if(len(visitedCities) >= countCities):
        visitedCities.clear()
        visitedCities.append(0)
    randomNumbers = list(set(saveState) - set(visitedCities))
    randomStep = random.choice(randomNumbers)
    visitedCities.append(randomStep)
    hypothesis.remove(randomStep)
    hypothesis.insert(cityIndex,randomStep)

# next city
def increment():
    global cityIndex
    global visitedCities
    if (cityIndex < countCities - 2):
        cityIndex += 1
    else:
        visitedCities.clear()
        cityIndex = 1

# calculates distance from tour
def getDistance(cities, hypothesis):
    distance = 0
    for i in range(countCities):
        if (i < countCities-1):
            distance += cities[hypothesis[i]][hypothesis[i+1]]
            print("[",hypothesis[i],"]",distance,"km ",end="")
        else:
            print("[",hypothesis[i],"]")

    return distance

if __name__ == '__main__':

    for i in range(countCities):
```

```
    hypothesis[i] = i
    for j in range(countCities):
        if (j > i):
            cities[i][j] = randint(1,100)
        elif(j < i):
            cities[i][j] = cities[j][i]

print("=== START ===");
while(lastFitness < threshold):

    print("_________________________________________________________")
    saveState = copy.deepcopy(hypothesis)
    doRandomStep()
    currentFitness = getFitness(lastFitness, hypothesis, saveState, cities)
    print("Old fitness ",lastFitness)
    print("Current fitness ",currentFitness)

    if (currentFitness > lastFitness):
        lastFitness = currentFitness
    elif(currentFitness < lastFitness):
        hypothesis = copy.deepcopy(saveState)
        if(trials < 3):
            increment()
        else:
            trials = 0
        visitedCities.append(saveState[cityIndex])
```

Output

```
PS D:\exp> python -u "d:\exp\exp3_uninformed.py"
=== START ===
____________________________________________________________
[ 0 ] 1.0 km [ 1 ] 94.0 km [ 2 ] 119.0 km [ 3 ] 188.0 km [ 4 ] 274.0 km [ 5 ] 334.0 km [ 6 ] 357.0 km [ 7 ] 425.0 km [ 8 ] 468.0 km [ 9 ] 562.0 km [ 10 ] 634.0 km
 [ 11 ] 725.0 km [ 12 ] 784.0 km [ 13 ] 861.0 km [ 14 ] 935.0 km [ 15 ] 999.0 km [ 16 ] 1015.0 km [ 17 ] 1016.0 km [ 18 ] 1019.0 km [ 19 ]
Old Distance  1019.0 km

[ 0 ] 53.0 km [ 15 ] 110.0 km [ 1 ] 203.0 km [ 2 ] 228.0 km [ 3 ] 297.0 km [ 4 ] 383.0 km [ 5 ] 443.0 km [ 6 ] 466.0 km [ 7 ] 534.0 km [ 8 ] 577.0 km [ 9 ] 671.0
km [ 10 ] 743.0 km [ 11 ] 834.0 km [ 12 ] 893.0 km [ 13 ] 970.0 km [ 14 ] 1001.0 km [ 16 ] 1017.0 km [ 17 ] 1018.0 km [ 18 ] 1021.0 km [ 19 ]
New Distance  1021.0 km

Old fitness  0
Current fitness  -1
____________________________________________________________
[ 0 ] 1.0 km [ 1 ] 94.0 km [ 2 ] 119.0 km [ 3 ] 188.0 km [ 4 ] 274.0 km [ 5 ] 334.0 km [ 6 ] 357.0 km [ 7 ] 425.0 km [ 8 ] 468.0 km [ 9 ] 562.0 km [ 10 ] 634.0 km
 [ 11 ] 725.0 km [ 12 ] 784.0 km [ 13 ] 861.0 km [ 14 ] 935.0 km [ 15 ] 999.0 km [ 16 ] 1015.0 km [ 17 ] 1016.0 km [ 18 ] 1019.0 km [ 19 ]
Old Distance  1019.0 km

[ 0 ] 1.0 km [ 1 ] 12.0 km [ 14 ] 73.0 km [ 2 ] 98.0 km [ 3 ] 167.0 km [ 4 ] 253.0 km [ 5 ] 313.0 km [ 6 ] 336.0 km [ 7 ] 404.0 km [ 8 ] 447.0 km [ 9 ] 541.0 km [
 10 ] 613.0 km [ 11 ] 704.0 km [ 12 ] 763.0 km [ 13 ] 848.0 km [ 15 ] 912.0 km [ 16 ] 928.0 km [ 17 ] 929.0 km [ 18 ] 932.0 km [ 19 ]
New Distance  932.0 km

Old fitness  0
Current fitness  1
```

```
____________________________________________________________
[ 0 ] 1.0 km [ 1 ] 12.0 km [ 14 ] 73.0 km [ 2 ] 98.0 km [ 3 ] 167.0 km [ 4 ] 253.0 km [ 5 ] 313.0 km [ 6 ] 336.0 km [ 7 ] 404.0 km [ 8 ] 447.0 km [ 9 ] 541.0 km [
 10 ] 613.0 km [ 11 ] 704.0 km [ 12 ] 763.0 km [ 13 ] 848.0 km [ 15 ] 912.0 km [ 16 ] 928.0 km [ 17 ] 929.0 km [ 18 ] 932.0 km [ 19 ]
Old Distance  932.0 km

[ 0 ] 1.0 km [ 1 ] 48.0 km [ 9 ] 66.0 km [ 14 ] 127.0 km [ 2 ] 152.0 km [ 3 ] 221.0 km [ 4 ] 307.0 km [ 5 ] 367.0 km [ 6 ] 390.0 km [ 7 ] 458.0 km [ 8 ] 461.0 km
[ 10 ] 533.0 km [ 11 ] 624.0 km [ 12 ] 683.0 km [ 13 ] 768.0 km [ 15 ] 832.0 km [ 16 ] 848.0 km [ 17 ] 849.0 km [ 18 ] 852.0 km [ 19 ]
New Distance  852.0 km

Old fitness  1
Current fitness  2
PS D:\exp>
```

Hill Climbing

Code:

```python
import numpy as np


def find_neighbours(state, landscape):
    neighbours = []
    dim = landscape.shape

    # left neighbour
    if state[0] != 0:
        neighbours.append((state[0] - 1, state[1]))

    # right neighbour
    if state[0] != dim[0] - 1:
        neighbours.append((state[0] + 1, state[1]))

    # top neighbour
    if state[1] != 0:
        neighbours.append((state[0], state[1] - 1))

    # bottom neighbour
    if state[1] != dim[1] - 1:
        neighbours.append((state[0], state[1] + 1))
```

```
    # top left
    if state[0] != 0 and state[1] != 0:
        neighbours.append((state[0] - 1, state[1] - 1))

    # bottom left
    if state[0] != 0 and state[1] != dim[1] - 1:
        neighbours.append((state[0] - 1, state[1] + 1))

    # top right
    if state[0] != dim[0] - 1 and state[1] != 0:
        neighbours.append((state[0] + 1, state[1] - 1))

    # bottom right
    if state[0] != dim[0] - 1 and state[1] != dim[1] - 1:
        neighbours.append((state[0] + 1, state[1] + 1))

    return neighbours


# Current optimization objective: local/global maximum
def hill_climb(curr_state, landscape):
    neighbours = find_neighbours(curr_state, landscape)
    bool
    ascended = False
    next_state = curr_state
    for neighbour in neighbours: #Find the neighbour with the greatest value
        if landscape[neighbour[0]][neighbour[1]] >
landscape[next_state[0]][next_state[1]]:
            next_state = neighbour
            ascended = True

    return ascended, next_state


def __main__():
    landscape = np.random.randint(1, high=50, size=(10, 10))
    print(landscape)
    start_state = (3, 6)  # matrix index coordinates
    current_state = start_state
    count = 1
    ascending = True
    while ascending:
        print("\nStep #", count)
        print("Current state coordinates: ", current_state)
```

```
        print("Current state value: ",
landscape[current_state[0]][current_state[1]])
        count += 1
        ascending, current_state = hill_climb(current_state, landscape)

    print("\nStep #", count)
    print("Optimization objective reached.")
    print("Final state coordinates: ", current_state)
    print("Final state value: ", landscape[current_state[0]][current_state[1]])


__main__()
```

Output

```
PS D:\exp> python -u "d:\exp\exp3_uninformed.py"
[[ 6 31 21 42 41  9 30 33 28 48]
 [ 4 47 41 31 11 24 29 10 39 41]
 [46  5 32 19 40 46 42 43  1 33]
 [31 36 28 22 27 45  7 36 40 14]
 [25  7  1 42 40 21 12 10 35 30]
 [28 17 40 43 27 42 10 25 47 24]
 [41 47  1 43 17 31 20 42 22 47]
 [33 20 39 37 42 35 46 20 43 45]
 [29 30 33 20  5 12 49 30 31 36]
 [18 31  9 28 11 41 49 20 21 27]]

Step # 1
Current state coordinates:  (3, 6)
Current state value:  7

Step # 2
Current state coordinates:  (2, 5)
Current state value:  46

Step # 3
Optimization objective reached.
Final state coordinates:  (2, 5)
Final state value:  46
PS D:\exp>
```

Exp 7 - Genetic Algorithm

Code

```
import random
def score(parent1, parent2):
   # doing crossover
   for i in range(len(parent1)-1, len(parent1)-4, -1):
```

```
        parent1[i], parent2[i] = parent2[i], parent1[i]
    #doint mutation by randomly selecting the genes
    mutation_index = [random.randint(0, len(parent1)-1) for i in
range(len(parent1)//2)]

    for i in mutation_index:
        if parent1[i] == '0':
            parent1[i] = '1'
        else:
            parent1[i] = '0'

        if parent2[i] == '0':
            parent2[i] = '1'
        else:
            parent2[i] = '0'

    score1 = parent1.count('1')
    score2 = parent2.count('1')
    #checking which child is better with more gene of type1
    if score1 > score2:
        return [''.join(parent1), score1]
    else:
        return [''.join(parent2), score2]

def genetic_algo():
    # Taking input as no. of parents
    n = int(input('Enter the number of parents: '))

    parents = []
    #taking parents genes as input 1 by 1

    for i in range(n):
        parents.append(list(input(f'Enter the parent{i+1}: ')))
    results = []

    #finding the score and storing it in results
    for i in range(len(parents)):
        for j in range(i+1, len(parents)):
            arr = [parents[i].copy(), parents[j].copy()]
            scores = score(parents[i], parents[j])
            results.append(scores + arr)

    # finding the best score among all combination of parents
    results.sort(key=lambda x: x[1], reverse=True)
```

```python
    print(f'The best offspring among the parents is : {results[0][0]} and the parents are {"".join(results[0][2])} and {"".join(results[0][3])}')

genetic_algo()
```

Output

```
PS D:\exp> python -u "d:\exp\exp3_uninformed.py"
Enter the number of parents: 4
Enter the parent1: 110101
Enter the parent2: 011011
Enter the parent3: 100001
Enter the parent4: 010011
The best offspring among the parents is : 011111 and the parents are 110101 and 011011
PS D:\exp>
```