[EXP 3 a - DLS](#_1mak5q6iqmib)

[EXP 3 a - IDDFS / DFIDS](#_gu2c9oo0sd2l)

[EXP 3 b - BFS (Best First Search)](#_2wfild96sgzi)

[EXP 3 b - A\*](#_oeb2hjzh3ahi)

[EXP 4 - Local Search (Hill Climbing)](#_89ldb24k1wpv)

[EXP 5 - Alpha Beta Pruning](#_tar1yhlohi8h)

[EXP 6 - CSP (Cryptarithmetic Problem)](#_gzslp9s7zk9f)

[EXP 7 - Genetic Algo (Knapsack Problem)](#_jlkforvfy91g)

[EXP 8 - Decision Tree](#_2cfk6ytmtklu)

[EXP 9 - Expert System](#_xet0imlwfezl)

| **EXP 3 a - DLS** |
| --- |
| class Node:  def \_\_init\_\_(self, value, children=None):  self.value = value  self.children = children if children else []  def depth\_limited\_search(node, goal, depth\_limit, current\_depth=0):  print(" " \* current\_depth + f"Searching node {node.value} at depth {current\_depth}/{depth\_limit}")    if node.value == goal.value:  return True    if current\_depth >= depth\_limit:  return False    for child in node.children:  if depth\_limited\_search(child, goal, depth\_limit, current\_depth + 1):  return True    return False  def print\_tree(node, indent=0, max\_depth=0):  if max\_depth >= indent:  print(" " \* indent + f"{node.value}")  for child in node.children:  print\_tree(child, indent + 1, max\_depth)  root = Node(1, [  Node(2, [Node(4, [Node(8), Node(9)]), Node(5, [Node(10), Node(11)])]),  Node(3, [Node(6, [Node(12), Node(13)]), Node(7, [Node(14), Node(15)])])  ])  goal\_node = root.children[0].children[1].children[1] # Node with value 11  depth\_limit = 2  found = depth\_limited\_search(root, goal\_node, depth\_limit)  print("\nTree structure:")  print\_tree(root, max\_depth=depth\_limit)  if found:  print("\nGoal node found within depth limit.")  else:  print("\nGoal node not found within depth limit.") |

| **EXP 3 a - IDDFS / DFIDS** |
| --- |
| from collections import defaultdict  class Graph:  def \_\_init\_\_(self, vertices):  self.V = vertices  self.graph = defaultdict(list)  def addEdge(self, u, v):  self.graph[u].append(v)  def DLS(self, src, target, maxDepth, currentDepth=0):  indent = " " \* currentDepth  print(f"{indent}Searching node {src} at depth {currentDepth}/{maxDepth}")  if src == target:  return True  if currentDepth >= maxDepth:  return False  for i in self.graph[src]:  if self.DLS(i, target, maxDepth, currentDepth + 1):  return True  return False  def IDDFS(self, src, target, maxDepth):  for i in range(maxDepth):  if self.DLS(src, target, i):  return True  return False  g = Graph(7)  g.addEdge(0, 1)  g.addEdge(0, 2)  g.addEdge(1, 3)  g.addEdge(1, 4)  g.addEdge(2, 5)  g.addEdge(2, 6)  target = 6  maxDepth = 3  src = 0  if g.IDDFS(src, target, maxDepth):  print("Goal node found within depth limit.")  else:  print("Goal node not found within depth limit") |

| **EXP 3 b - BFS (Best First Search)** |
| --- |
| from queue import PriorityQueue  # Define a sample graph as an adjacency list  graph = {  'S': [('A', 3), ('B', 6), ('C', 5)],  'A': [('D', 9), ('E', 8)],  'B': [('F', 12), ('G', 14)],  'C': [('H', 7)],  'D': [],  'E': [],  'F': [],  'G': [],  'H': [('I', 5), ('J', 6)],  'I': [('K', 1), ('L', 10), ('M', 2)],  'J': [],  'K': [],  'L': [],  'M': []  }  def best\_first\_search(graph, start, goal):  visited = set()  priority\_queue = PriorityQueue()  priority\_queue.put((0, start))  while not priority\_queue.empty():  cost, node = priority\_queue.get()  print(f"Exploring node: {node} (Cost: {cost})")  if node == goal:  return True  if node not in visited:  visited.add(node)  for neighbor, neighbor\_cost in graph[node]:  if neighbor not in visited:  priority\_queue.put((neighbor\_cost, neighbor))  # print("Priority Queue after exploring node:")  # for item in priority\_queue.queue:  # print(f" Node: {item[1]}, Cost: {item[0]}")  return False  start\_node = 'S'  goal\_node = 'I'  if best\_first\_search(graph, start\_node, goal\_node):  print(f"Goal node '{goal\_node}' found.")  else:  print(f"Goal node '{goal\_node}' not found.") |

| **EXP 3 b - A\*** |
| --- |
| 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]  def h(self, n):  H = {  'S': 1, 'A': 1, 'B': 1, 'C': 1, 'D': 1, 'E': 1, 'F': 1, 'G': 1, 'H': 1, 'I': 1, 'J': 1, 'K': 1, 'L': 1, 'M': 1  }  return H[n]  def a\_star\_algorithm(self, start\_node, stop\_node):  open\_list = set([start\_node])  closed\_list = set([])  g = {}  g[start\_node] = 0  parents = {}  parents[start\_node] = start\_node  while len(open\_list) > 0:  n = None  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 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 (m, weight) in self.get\_neighbors(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)  open\_list.remove(n)  closed\_list.add(n)  print('Path does not exist!')  return None    adjacency\_list = {  'S': [('A', 3), ('B', 6), ('C', 5)],  'A': [('D', 9), ('E', 8)],  'B': [('F', 12), ('G', 14)],  'C': [('H', 7)],  'D': [],  'E': [],  'F': [],  'G': [],  'H': [('I', 5), ('J', 6)],  'I': [('K', 1), ('L', 10), ('M', 2)],  'J': [],  'K': [],  'L': [],  'M': []  }  graph1 = Graph(adjacency\_list)  graph1.a\_star\_algorithm('S', 'I') |

| **EXP 4 - Local Search (Hill Climbing)** |
| --- |
| import random  import numpy as np  import networkx as nx  #coordinate of the points/cities  coordinate = np.array([[1,2], [30,21], [56,23], [8,18], [20,50], [3,4], [11,6], [6,7], [15,20], [10,9],  [12,12]])  #adjacency matrix for a weighted graph based on the given coordinates  def generate\_matrix(coordinate):  matrix = []  for i in range(len(coordinate)):  for j in range(len(coordinate)) :  p = np.linalg.norm(coordinate[i] - coordinate[j])  matrix.append(p)  matrix = np.reshape(matrix, (len(coordinate),len(coordinate)))  #print(matrix)  return matrix  #finds a random solution  def solution(matrix):  points = list(range(0, len(matrix)))  solution = []  for i in range(0, len(matrix)):  random\_point = points[random.randint(0, len(points) - 1)]  solution.append(random\_point)  points.remove(random\_point)  return solution  #calculate the path based on the random solution  def path\_length(matrix, solution):  cycle\_length = 0  for i in range(0, len(solution)):  cycle\_length += matrix[solution[i]][solution[i - 1]]  return cycle\_length  #generate neighbors of the random solution by swapping cities and returns the best neighbor  def neighbors(matrix, solution):  neighbors = []  for i in range(len(solution)):  for j in range(i + 1, len(solution)):  neighbor = solution.copy()  neighbor[i] = solution[j]  neighbor[j] = solution[i]  neighbors.append(neighbor)    #assume that the first neighbor in the list is the best neighbor  best\_neighbor = neighbors[0]  best\_path = path\_length(matrix, best\_neighbor)    #check if there is a better neighbor  for neighbor in neighbors:  current\_path = path\_length(matrix, neighbor)  if current\_path < best\_path:  best\_path = current\_path  best\_neighbor = neighbor  return best\_neighbor, best\_path  def hill\_climbing(coordinate):  matrix = generate\_matrix(coordinate)    current\_solution = solution(matrix)  current\_path = path\_length(matrix, current\_solution)  neighbor = neighbors(matrix,current\_solution)[0]  best\_neighbor, best\_neighbor\_path = neighbors(matrix, neighbor)  while best\_neighbor\_path < current\_path:  current\_solution = best\_neighbor  current\_path = best\_neighbor\_path  neighbor = neighbors(matrix, current\_solution)[0]  best\_neighbor, best\_neighbor\_path = neighbors(matrix, neighbor)  return current\_path, current\_solution  final\_solution = hill\_climbing(coordinate)  print("The solution is \n", final\_solution[1]) |

| **EXP 5 - Alpha Beta Pruning** |
| --- |
| # 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  # Driver Code  if \_\_name\_\_ == "\_\_main\_\_":  values = [3, 5, 6, 9, 1, 2, 0, -1]  print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX)) |

| **EXP 6 - CSP (Cryptarithmetic Problem)** |
| --- |
| class CSP\_crypt:  def \_\_init\_\_(self, str):  p = str.split()  self.p1 = p[0]  self.p2 = p[2]  self.p3 = p[4]  self.opr = p[1]  self.state = []  self.solved = False  for q in self.p1:  if not (q in self.state):  self.state.append(q)  for q in self.p2:  if not (q in self.state):  self.state.append(q)  for q in self.p3:  if not (q in self.state):  self.state.append(q)  for i in range(10 - len(self.state)):  self.state.append('x')  def display(self):  print("Line 1: ", self.p1)  print("Line 2: ", self.p2)  print("Line 3: ", self.p3)  print("Operation: ", self.opr)  print("State: ", self.state)  print("Solved: ", self.solved)  def display\_ans(self):  for i in self.state:  if not (i == 'x'):  print(i, " - ", self.state.index(i))  def apply\_constraints(self, depth):  if (len(self.p3) > len(self.p1)) and (len(self.p3) > len(self.p2)):  if self.state[0] == self.p3[0]:  return True  if self.state[1] == self.p3[0]:  return True  elif depth < 2:  return True  else:  return True  return False  def get\_number(self, p):  num = 0  for q in p:  num = num \* 10  num = num + self.state.index(q)  return num  def solve(self):  num1 = self.get\_number(self.p1)  num2 = self.get\_number(self.p2)  num3 = self.get\_number(self.p3)  if self.opr == '+':  ans = num1 + num2  elif self.opr == '-':  ans = num1 - num2  elif self.opr == '\*':  ans = num1 \* num2  elif self.opr == '/':  ans = num1 / num2  if ans == num3:  print("ans = ", ans)  print("num1 = ", num1)  print("num2 = ", num2)  print("num3 = ", num3)  self.solved = True  def expand(self, l, r, depth):  self.solve()  if self.solved:  return  elif l == r:  return  else:  for i in range(l, r + 1):  self.state[l], self.state[i] = self.state[i], self.state[l]  if self.apply\_constraints(depth):  depth = depth + 1  self.expand(l + 1, r, depth)  depth = depth - 1  if self.solved:  return  self.state[i], self.state[l] = self.state[l], self.state[i]  if \_\_name\_\_ == "\_\_main\_\_":  str = input("Enter the problem: ")  c\_csp = CSP\_crypt(str)  c\_csp.display()  c\_csp.expand(0, 9, 0)  c\_csp.display()  if c\_csp.solved:  c\_csp.display\_ans() |

| **EXP 7 - Genetic Algo (Knapsack Problem)** |
| --- |
| import random  # function to generate a random population  def generate\_population(size):  population = []  for \_ in range(size):  genes = [0, 1]  chromosome = []  for \_ in range(len(items)):  chromosome.append(random.choice(genes))  population.append(chromosome)  print("Generated a random population of size", size)  return population  # function to calculate the fitness of a chromosome  def calculate\_fitness(chromosome):  total\_weight = 0  total\_value = 0  for i in range(len(chromosome)):  if chromosome[i] == 1:  total\_weight += items[i][0]  total\_value += items[i][1]  if total\_weight > max\_weight:  return 0  else:  return total\_value  # function to select two chromosomes for crossover  def select\_chromosomes(population):  fitness\_values = []  for chromosome in population:  fitness\_values.append(calculate\_fitness(chromosome))    fitness\_values = [float(i)/sum(fitness\_values) for i in fitness\_values]    parent1 = random.choices(population, weights=fitness\_values, k=1)[0]  parent2 = random.choices(population, weights=fitness\_values, k=1)[0]    print("Selected two chromosomes for crossover")  return parent1, parent2  # function to perform crossover between two chromosomes  def crossover(parent1, parent2):  crossover\_point = random.randint(0, len(items)-1)  child1 = parent1[0:crossover\_point] + parent2[crossover\_point:]  child2 = parent2[0:crossover\_point] + parent1[crossover\_point:]    print("Performed crossover between two chromosomes")  return child1, child2  # function to perform mutation on a chromosome  def mutate(chromosome):  mutation\_point = random.randint(0, len(items)-1)  if chromosome[mutation\_point] == 0:  chromosome[mutation\_point] = 1  else:  chromosome[mutation\_point] = 0  print("Performed mutation on a chromosome")  return chromosome  # function to get the best chromosome from the population  def get\_best(population):  fitness\_values = []  for chromosome in population:  fitness\_values.append(calculate\_fitness(chromosome))  max\_value = max(fitness\_values)  max\_index = fitness\_values.index(max\_value)  return population[max\_index]  # items that can be put in the knapsack  items = [  [1, 2],  [2, 4],  [3, 4],  [4, 5],  [5, 7],  [6, 9]  ]  # print available items  print("Available items:\n", items)  # parameters for genetic algorithm  max\_weight = 10  population\_size = 10  mutation\_probability = 0.2  generations = 10  print("\nGenetic algorithm parameters:")  print("Max weight:", max\_weight)  print("Population:", population\_size)  print("Mutation probability:", mutation\_probability)  print("Generations:", generations, "\n")  print("Performing genetic evolution:")  # generate a random population  population = generate\_population(population\_size)  # evolve the population for specified number of generations  for \_ in range(generations):  # select two chromosomes for crossover  parent1, parent2 = select\_chromosomes(population)  # perform crossover to generate two new chromosomes  child1, child2 = crossover(parent1, parent2)  # perform mutation on the two new chromosomes  if random.uniform(0, 1) < mutation\_probability:  child1 = mutate(child1)  if random.uniform(0, 1) < mutation\_probability:  child2 = mutate(child2)  # replace the old population with the new population  population = [child1, child2] + population[2:]  # get the best chromosome from the population  best = get\_best(population)  # get the weight and value of the best solution  total\_weight = 0  total\_value = 0  for i in range(len(best)):  if best[i] == 1:  total\_weight += items[i][0]  total\_value += items[i][1]  # print the best solution  print("\nThe best solution:")  print("Weight:", total\_weight)  print("Value:", total\_value) |

| **EXP 8 - Decision Tree** |
| --- |
| class Node:  def \_\_init\_\_(self, value, children=None):  self.value = value  self.children = children if children else []  def depth\_limited\_search(node, goal, depth\_limit, current\_depth=0):  print(" " \* current\_depth + f"Searching node {node.value} at depth {current\_depth}/{depth\_limit}")    if node.value == goal.value:  return True    if current\_depth >= depth\_limit:  return False    for child in node.children:  if depth\_limited\_search(child, goal, depth\_limit, current\_depth + 1):  return True    return False  def print\_tree(node, indent=0, max\_depth=0):  if max\_depth >= indent:  print(" " \* indent + f"{node.value}")  for child in node.children:  print\_tree(child, indent + 1, max\_depth)  root = Node(1, [  Node(2, [Node(4, [Node(8), Node(9)]), Node(5, [Node(10), Node(11)])]),  Node(3, [Node(6, [Node(12), Node(13)]), Node(7, [Node(14), Node(15)])])  ])  goal\_node = root.children[0].children[1].children[1] # Node with value 11  depth\_limit = 2  found = depth\_limited\_search(root, goal\_node, depth\_limit)  print("\nTree structure:")  print\_tree(root, max\_depth=depth\_limit)  if found:  print("\nGoal node found within depth limit.")  else:  print("\nGoal node not found within depth limit.") |

| **EXP 9 - Expert System** |
| --- |
| class DietExpertSystem:  def \_\_init\_\_(self):  # Define knowledge base (diet rules)  self.knowledge\_base = {  "Vegetarian": "You should focus on plant-based foods and avoid meat and animal products.",  "Vegan": "You should follow a strict plant-based diet, avoiding all animal products.",  "Omnivore": "You have a wide range of food options, including both plant-based and animal-based foods.",  }  def get\_diet\_recommendation(self, dietary\_preference):  # Check if the dietary preference is defined in the knowledge base  recommendation = self.knowledge\_base.get(dietary\_preference)  if recommendation:  return recommendation  else:  return "I don't have a specific recommendation for that dietary preference."  # Create an instance of the DietExpertSystem  diet\_system = DietExpertSystem()  # Example usage:  dietary\_preference = "Vegetarian"  recommendation = diet\_system.get\_diet\_recommendation(dietary\_preference)  print(f"If you are a {dietary\_preference}, {recommendation}") |