```python
#dfs path finding algorithm
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start)
    for next in graph[start] - visited:
        dfs(graph, next, visited)
    return visited
graph = {'0': set(['1', '2']),
         '1': set(['0', '3', '4']),
         '2': set(['0']),
         '3': set(['1']),
         '4': set(['2', '3'])}
dfs(graph, '0')
```

```
0
2
1
3
4
{'0', '1', '2', '3', '4'}
```

```python
#bfs for shortest path
import collections
def bfs(graph, root):
    visited, queue = set(), collections.deque([root])
    visited.add(root)
    while queue:
        vertex = queue.popleft()
        print(str(vertex) + " ", end="")
        for neighbour in graph[vertex]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)
if __name__ == '__main__':
    graph = {0: [1, 2], 1: [2], 2: [3], 3: [1, 2]}
    print("Following is Breadth First Traversal: ")
    bfs(graph, 0)
```

```
Following is Breadth First Traversal:
0 1 2 3
```

```
#a*
from sys import maxsize
from itertools import permutations
V = 4
def travellingSalesmanProblem(graph, s):
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)
    min_path = maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:
        current_pathweight = 0
        k = s
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][s]
        min_path = min(min_path, current_pathweight)
    return min_path
if __name__ == "__main__":
    graph = [[0, 10, 15, 20], [10, 0, 35, 25],
            [15, 35, 0, 30], [20, 25, 30, 0]]
    s = 0
    print(travellingSalesmanProblem(graph, s))
```

```
80
```

```
#a* tsp
from collections import deque
class Graph:
    def __init__(self, adjac_lis):
        self.adjac_lis = adjac_lis
    def get_neighbors(self, v):
        return self.adjac_lis[v]
    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }
        return H[n]
    def a_star_algorithm(self, start, stop):
        open_lst = set([start])
        closed_lst = set([])
        poo = {}
        poo[start] = 0
        par = {}
        par[start] = start
        while len(open_lst) > 0:
            n = None
            for v in open_lst:
                if n == None or poo[v] + self.h(v) < poo[n] + self.h(n):
                    n = v;
            if n == None:
                print('Path does not exist!')
                return None
            if n == stop:
                reconst_path = []
                while par[n] != n:
                    reconst_path.append(n)
                    n = par[n]
                reconst_path.append(start)
                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_lst and m not in closed_lst:
                    open_lst.add(m)
                    par[m] = n
                    poo[m] = poo[n] + weight
                else:
                    if poo[m] > poo[n] + weight:
                        poo[m] = poo[n] + weight
                        par[m] = n
                        if m in closed_lst:
                            closed_lst.remove(m)
                            open_lst.add(m)
            open_lst.remove(n)
            closed_lst.add(n)
        print('Path does not exist!')
        return None
adjac_lis = {
  'A': [('B', 1), ('C', 3), ('D', 7)],
  'B': [('D', 5)],
  'C': [('D', 12)]
}
graph1 = Graph(adjac_lis)
graph1.a_star_algorithm('A', 'D')
```

```
Path found: ['A', 'B', 'D']
['A', 'B', 'D']
```

```python
#minimax
import math
def minimax (curDepth, nodeIndex,
             maxTurn, scores,
             targetDepth):
    if (curDepth == targetDepth):
        return scores[nodeIndex]
    if (maxTurn):
        return max(minimax(curDepth + 1, nodeIndex * 2,
                    False, scores, targetDepth),
                   minimax(curDepth + 1, nodeIndex * 2 + 1,
                    False, scores, targetDepth))

    else:
        return min(minimax(curDepth + 1, nodeIndex * 2,
                     True, scores, targetDepth),
                   minimax(curDepth + 1, nodeIndex * 2 + 1,
                     True, scores, targetDepth))
scores = [3, 5, 2, 9, 12, 5, 23, 23]
treeDepth = math.log(len(scores), 2)
print("The optimal value is : ", end = "")
print(minimax(0, 0, True, scores, treeDepth))
```

```
The optimal value is : 12
```

```python
#csp (graph colour)
class Graph():
  def __init__(self, vertices):
    self.V = vertices
    self.graph = [[0 for column in range(vertices)]\
              for row in range(vertices)]
  def isSafe(self, v, colour, c):
    for i in range(self.V):
      if self.graph[v][i] == 1 and colour[i] == c:
        return False
    return True
  def graphColourUtil(self, m, colour, v):
    if v == self.V:
      return True
    for c in range(1, m + 1):
      if self.isSafe(v, colour, c) == True:
        colour[v] = c
        if self.graphColourUtil(m, colour, v + 1) == True:
          return True
        colour[v] = 0
  def graphColouring(self, m):
    colour = [0] * self.V
    if self.graphColourUtil(m, colour, 0) == None:
      return False
    print ("Solution exist and Following are the assigned colours:")
    for c in colour:
      print (c,end=' ')
    return True
g = Graph(4)
g.graph = [[0, 1, 1, 1], [1, 0, 1, 0], [1, 1, 0, 1], [1, 0, 1, 0]]
m = 3
g.graphColouring(m)
```

```
Solution exist and Following are the assigned colours:
1 2 3 2 True
```

```
#dfs topological
from collections import defaultdict
class Graph:
  def __init__(self, vertices):
    self.graph = defaultdict(list)
    self.V = vertices
  def addEdge(self, u, v):
    self.graph[u].append(v)
  def topologicalSortUtil(self, v, visited, stack):
    visited[v] = True
    for i in self.graph[v]:
      if visited[i] == False:
        self.topologicalSortUtil(i, visited, stack)
    stack.append(v)
  def topologicalSort(self):
    visited = [False]*self.V
    stack = []
    for i in range(self.V):
      if visited[i] == False:
        self.topologicalSortUtil(i, visited, stack)
    print(stack[::-1])
g = Graph(6)
g.addEdge(5, 2)
g.addEdge(5, 0)
g.addEdge(4, 0)
g.addEdge(4, 1)
g.addEdge(2, 3)
g.addEdge(3, 1)
print ("Following is a Topological Sort of the given graph")
```

```
g.topologicalSort()
```

```
Following is a Topological Sort of the given graph
[5, 4, 2, 3, 1, 0]
```

```
#Implement alpha-beta tree search with pruning
MAX, MIN = 1000, -1000
def minimax(depth, nodeIndex, maximizingPlayer,
      values, alpha, beta):
  if depth == 3:
    return values[nodeIndex]
  if maximizingPlayer:
    best = MIN
    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)
      if beta <= alpha:
        break
    return best
  else:
    best = MAX
    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)
      if beta <= alpha:
        break
    return best
if __name__ == "__main__":
  values = [3, 5, 6, 9, 1, 2, 0, -1]
  print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

```
The optimal value is : 5
```

```
#Implement backtracking algorithm using CSP for 4-Queen problem
global N
N = 4
def printSolution(board):
  for i in range(N):
    for j in range(N):
      print (board[i][j], end = " ")
    print()
```

```
def isSafe(board, row, col):
  for i in range(col):
    if board[row][i] == 1:
      return False
  for i, j in zip(range(row, -1, -1),
          range(col, -1, -1)):
    if board[i][j] == 1:
      return False
  for i, j in zip(range(row, N, 1),
          range(col, -1, -1)):
    if board[i][j] == 1:
      return False
  return True
def solveNQUtil(board, col):
  if col >= N:
    return True
  for i in range(N):
    if isSafe(board, i, col):
      board[i][col] = 1
      if solveNQUtil(board, col + 1) == True:
        return True
      board[i][col] = 0
  return False
def solveNQ():
  board = [ [0, 0, 0, 0],
      [0, 0, 0, 0],
      [0, 0, 0, 0],
      [0, 0, 0, 0] ]
  if solveNQUtil(board, 0) == False:
    print ("Solution does not exist")
    return False
  printSolution(board)
  return True
solveNQ()
```

```
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
True
```

```
#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()
```

```
([2, 1, 0, 3], 1400)
```