8 puzzle

```
import heapq

# Goal state for the 8-puzzle problem
goal_state = [[1, 2, 3],
              [4, 5, 6],
              [7, 8, 0]]

# Movements: up, down, left, right
movements = [(-1, 0), (1, 0), (0, -1), (0, 1)]

class Node:
   def __init__(self, state, parent, move, depth, cost):
      self.state = state
      self.parent = parent
      self.move = move
      self.depth = depth
      self.cost = cost

   def __lt__(self, other):
      return self.cost < other.cost

def a_star(start_state):
   open_list = []
   closed_list = set()
   start_node = Node(start_state, None, None, 0, heuristic(start_state))
   heapq.heappush(open_list, start_node)

   while open_list:
      current_node = heapq.heappop(open_list)
      closed_list.add(tuple(map(tuple, current_node.state)))
```

```
        if current_node.state == goal_state:
            return reconstruct_path(current_node)

        for move in movements:
            new_state = make_move(current_node.state, move)
            if new_state is None:
                continue

            new_node = Node(new_state, current_node, move, current_node.depth + 1,
current_node.depth + 1 + heuristic(new_state))

            if tuple(map(tuple, new_state)) not in closed_list:
                heapq.heappush(open_list, new_node)

    return None

def heuristic(state):
    cost = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                x, y = divmod(state[i][j] - 1, 3)
                cost += abs(x - i) + abs(y - j)
    return cost

def make_move(state, move):
    new_state = [row[:] for row in state]
    x, y = next((i, j) for i in range(3) for j in range(3) if new_state[i][j] == 0)
    dx, dy = move
    nx, ny = x + dx, y + dy
```

```
    if 0 <= nx < 3 and 0 <= ny < 3:
        new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
        return new_state
    return None

def reconstruct_path(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return path[::-1]

def print_state(state):
    for row in state:
        print(' '.join(str(tile) if tile != 0 else ' ' for tile in row))
    print()

# Initial state for the 8-puzzle problem
initial_state = [[1, 2, 3],
                 [4, 0, 5],
                 [6, 7, 8]]

solution = a_star(initial_state)
if solution:
    print("Solution found:")
    for step in solution:
        print_state(step)
else:
    print("No solution found")
```

8 quuen

```
N=8
def solveNqueens(board,col):
    if col==N:
        print(board)
        return True
    for i in range(N):
        if isSafe(board,i,col):
            board[i][col]=1
            if solveNqueens(board,col+1):
                return True
            board[i][col]=0
    return False
def isSafe(board,row,col):
    for x in range(col):
        if board[row][x]==1:
            return False

    for x,y in zip(range(row,-1,-1),range(col,-1,-1)):
        if board[x][y]==1:
            return False

    for x,y in zip(range(row,N,1),range(col,-1,-1)):
        if board[x][y]==1:
            return False

    return True
board=[[0 for x in range(N)]for y in range(N)]
if not solveNqueens(board,0):
    print("no sotion found")
```

cript arithematic

```
import itertools
def is_valid_solution(b,a,s,e,l,g,m):
    base=b*1000+a*100+s*10+e
    ball=b*1000+a*100+l*10+l
    games=g*10000+a*1000+m*100+e*10+s
    return base+ball==games
digits=range(10)
for perm in itertools.permutations(digits,7):
    b,a,s,e,l,g,m=perm
    if b!=0 and is_valid_solution(b,a,s,e,l,g,m):
        print(f"BASE={b}{a}{s}{e}")
        print(f"BALL={b}{a}{l}{l}")
        print(f"GAMES={g}{a}{m}{e}{s}")
        break
```

vaccume cleaner

```
from collections import deque
movements=[(-1,0),(1,0),(0,-1),(0,1)]
def is_valid_move(x,y,grid):
    rows,cols=len(grid),len(grid[0])
    return 0<=x<rows  and 0<=y<cols and grid[x][y]!='#'
def bfs_vaccume_cleaner(start,grid):
    rows,cols=len(grid),len(grid[0])
    queue=deque([(start,0)])
    visited=set()
    visited.add(start)
    while queue:
        (x,y),steps=queue.popleft()
        if grid[x][y]=='D':
            grid[x][y]='C'
```

```
            print(f"cleaned:({x},{y}) in {steps} steps")
        if all(grid[i][j]!='D' for i in range(rows) for j in range (cols)):
            print("all spots are cleaned")
            return steps
        for dx,dy in movements:
            nx,ny=x+dx,y+dy
            if is_valid_move(nx,ny,grid) and (nx,ny) not in visited:
                visited.add((nx,ny))
                queue.append(((nx,ny),steps+1))

    print("some spots could not cleaned")
    return -1
def print_grid(grid):
    for row in grid:
        print(" ".join(row))
    print()
grid=[['D','D','D'],
      ['D','#','D'],
      ['D','D','D']]
start=(1,0)
print("initial grid:")
print_grid(grid)

steps=bfs_vaccume_cleaner(start,grid)

print("\n final state:")
print_grid(grid)
print(f"total steps={steps}")
```

bfs

```
from collections import deque
def bfs(graph,start_node):
    visited=set()
    queue=deque([start_node])
    visited.add(start_node)
    while queue:
        node=queue.popleft()
        print(node,end=" ")

        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['A','D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
```

Dfs

```
from collections import deque
def dfs(graph, start_node):
    visited = set()
    stack = [start_node]

    while stack:
```

```
        node = stack.pop()
        if node not in visited:
            print(node, end=" ")
            visited.add(node)

            for neighbor in reversed(graph[node]):
                if neighbor not in visited:
                    stack.append(neighbor)

# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

print("DFS traversal starting from node 'A':")
dfs(graph, 'A')
```

tsp

```
import itertools
def distance(city1,city2,distance_matrix):
    return distance_matrix[city1][city2]
def total_distance(route,distance_matrix):
    total_dist=0
    for i in range(len(route)-1):
        total_dist+=distance(route[i],route[i+1],distance_matrix)
    total_dist+=distance(route[-1],route[0],distance_matrix)
```

```
    return total_dist
def travelling_salesman_problem(cities,distance_matrix):
    shortest_route=None
    min_distance=float('inf')
    for perm in itertools.permutations(cities):
        current_distance=total_distance(perm,distance_matrix)
        if current_distance<min_distance:
            min_distance=current_distance
            shortest_route=perm
    return shortest_route,min_distance
cities=[0,1,2,3]
distance_matrix=[
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]
route,distance=travelling_salesman_problem(cities,distance_matrix)
print(f"shortest route={route}")
print(f"distance={distance}")`
```

minmax

```
import math
def minmax(depth,index,maxturn,scores,targetdepth):
    if(depth==targetdepth):
        return scores[index]
    if(maxturn):
        return
max(minmax(depth+1,index*2,False,scores,targetdepth),minmax(depth+1,index*2+1,False,scores,tar
getdepth))
    else:
```

```
    return
min(minmax(depth+1,index*2,True,scores,targetdepth),minmax(depth+1,index*2+1,True,scores,targ
etdepth))

scores=[3,5,2,9,12,5,23,23]

treedepth=math.log(len(scores),2)

print("optimal value=",end=" ")

print(minmax(0,0,True,scores,treedepth))
```

alpha beta

```
def minmax(depth,index,maxturn,scores,alpha,beta):
  if depth==3:
    return scores[index]
  if maxturn:
    max_eval=float('-inf')
    for i in range(2):
      eval=minmax(depth+1,index*2+i,False,scores,alpha,beta)
      max_eval=max(max_eval,eval)
      alpha=max(alpha,eval)
      if alpha>=beta:
        break
    return max_eval
  else:
    min_eval=float('inf')
    for i in range(2):
      eval=minmax(depth+1,index*2+i,True,scores,alpha,beta)
      min_eval=min(min_eval,eval)
      beta=min(beta,eval)
      if alpha>=beta:
        break
    return min_eval
if __name__=="__main__":
  scores=[3,5,6,9,1,2,0,-1]
```

```
    print("optimal value:",minmax(0,0,True,scores,float('-inf'),float('inf')))
```

a* search

```
import heapq
def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
def greedy_best_first_search(mat, start, end):
    open_list = []
    came_from = {}
    visited = set()
    heapq.heappush(open_list, (heuristic(start, end), start))
    came_from[start] = None
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
    while open_list:
        current_heuristic, current = heapq.heappop(open_list)
        if current == end:
            path = []
            while current:
                path.append(current)
                current = came_from[current]
            return path[::-1]
        visited.add(current)
        for direction in directions:
            neighbor = (current[0] + direction[0], current[1] + direction[1])
            if (0 <= neighbor[0] < len(mat)) and (0 <= neighbor[1] < len(mat[0])):
                if mat[neighbor[0]][neighbor[1]] == 0 and neighbor not in visited:
                    visited.add(neighbor)
                    came_from[neighbor] = current
                    heapq.heappush(open_list, (heuristic(neighbor, end), neighbor))
    return None
mat = [[0, 0, 0, 0, 1],
```

```
    [0, 1, 1, 0, 1],
    [0, 0, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 0, 0]]
start = (0, 0)
end = (4, 4)
path = greedy_best_first_search(mat, start, end)
print("Path from start to end:", path)
```

Prolog

17

```
sum_to_n(0, 0).
sum_to_n(N, Sum) :-
    N > 0,
    N1 is N - 1,
    sum_to_n(N1, Sum1),
    Sum is N + Sum1.
```

18(dob)

```
person(swetha,date(2004,09,29)).
person(harshi,date(2008,05,10)).
dob(Name,DOB):-
    person(Name,DOB).
```

19(student teacher)

```
student(sai, csa1732).
teacher(kumar, csa1732).
subject_code(csa1732, 'AI').
student_subject(Student, SubjectCode) :-
    student(Student, SubjectCode).
```

```
student_teacher(Student, Teacher) :-
    student(Student, SubjectCode), teacher(Teacher, SubjectCode).
teacher_subject(Teacher, SubjectCode) :-
    teacher(Teacher, SubjectCode).
subject_name(SubjectCode, SubjectName) :-
    subject_code(SubjectCode, SubjectName).
student_subject_name(Student, SubjectName) :-
    student(Student, SubjectCode), subject_code(SubjectCode, SubjectName).
student_teacher_subject(Student, Teacher, SubjectName) :-
    student(Student, SubjectCode), teacher(Teacher, SubjectCode), subject_code(SubjectCode, 
SubjectName).
```

Or

```
studies(teja,csa1732,komali,ai).
studies(varsha,csa0972,pandu,java).
details(Student,Course,Teacher,Subject):-
    studies(Student,Course,Teacher,Subject).
class(Student,Subject):-
    studies(Student,,,Subject).
```

20(planet)

```
planet(mercury, rocky, small, closest_to_sun).
planet(venus, rocky, medium, second_closest_to_sun).
planet(earth, rocky, medium, third_closest_to_sun).
planet(mars, rocky, small, fourth_closest_to_sun).
planet(jupiter, gas_giant, large, fifth_closest_to_sun).
planet(saturn, gas_giant, large, sixth_closest_to_sun).
planet(uranus, ice_giant, medium, seventh_closest_to_sun).
planet(neptune, ice_giant, medium, eighth_closest_to_sun).

% Rules
% Finding planets by type
```

```
planet_type(Name, Type) :-
   planet(Name, Type, _, _).

% Finding planets by size
planet_size(Name, Size) :-
   planet(Name, _, Size, _).

% Finding planets by position from the sun
planet_position(Name, Position) :-
   planet(Name, _, _, Position).
```

21.towers of Hanoi

```
% Define predicate to solve Towers of Hanoi
hanoi(N) :-
   move(N, left, center, right).

% Base case: Moving 0 discs requires no moves
move(0, _, _, _) :- !.

% Recursive case: Move N discs from A to C using B as auxiliary
move(N, A, B, C) :-
   N > 0,
   M is N - 1,
   move(M, A, C, B),    % Move N-1 discs from A to B using C
   move_disk(A, C),     % Move the Nth disc from A to C
   move(M, B, A, C).    % Move N-1 discs from B to C using A

% Helper predicate to print the move
move_disk(From, To) :-
   format('Move disk from ~w to ~w~n', [From, To]).
```

22(BIRD)

% Facts about specific birds

bird(sparrow).

bird(penguin).

bird(ostrich).

bird(eagle).

% Facts about birds that cannot fly

cannot_fly(penguin).

cannot_fly(ostrich).

% General rule: birds can fly unless specified otherwise or if they are injured

can_fly(X) :- bird(X), \+ cannot_fly(X), \+ injured(X).

% Example of an injured bird

injured(sparrow). % You can comment this out to see the change in output

% To check if a bird can fly

is_flying(Bird) :- can_fly(Bird), write(Bird), write(' can fly.'), nl.

is_flying(Bird) :- \+ can_fly(Bird), write(Bird), write(' cannot fly.'), nl.

23(family tree)

parent(john, mary).

parent(john, david).

parent(mary, susan).

parent(david, tom).

parent(david, anna).

male(john).

```
male(david).
male(tom).

female(mary).
female(susan).
female(anna).

% Rules
father(F, C) :- parent(F, C), male(F).
mother(M, C) :- parent(M, C), female(M).
sibling(X, Y) :- parent(P, X), parent(P, Y), X \= Y.
% Grandparent relationship
grandparent(GP, GC) :- parent(GP, P), parent(P, GC).

% Granddaughter relationship
granddaughter(GD, GP) :- grandparent(GP, GD), female(GD).
grandson(GS,GP):- grandparent(GP,GS),male(GS).
```

24(diet)

```
diet(diabetes, 'Low sugar, high fiber, whole grains, vegetables, lean protein').
diet(hypertension, 'Low sodium, high potassium, fruits, vegetables, whole grains').
diet(heart_disease, 'Low saturated fat, high omega-3, fruits, vegetables, whole grains').
diet(obesity, 'Balanced diet, portion control, high fiber, lean protein').
diet(anemia, 'High iron, vitamin C, leafy greens, red meat, beans').
diet(gastrointestinal_disorder, 'Low fiber, bland diet, avoid spicy foods, small frequent meals').
% Rules
suggest_diet(Disease, Diet) :- diet(Disease, Diet).
```

25.Monkey

```
% Define the initial state
initial_state(state(at_door, on_floor, at_window, has_not)).
```

```
% Define the goal state
goal_state(state(_, _, _, has)).

% Define the possible actions
action(state(middle, on_box, middle, has_not), grasp, state(middle, on_box, middle, has)).
action(state(P, on_floor, P, H), climb_box, state(P, on_box, P, H)).
action(state(P1, on_floor, P1, H), push_box(P1, P2), state(P2, on_floor, P2, H)).
action(state(P1, on_floor, B, H), walk(P1, P2), state(P2, on_floor, B, H)).

% Define a plan to achieve the goal state
plan(State, [], State) :- goal_state(State).
plan(State1, [Action | RestActions], State3) :-
    action(State1, Action, State2),
    plan(State2, RestActions, State3).

% Query to find the plan
find_plan(Plan) :-
    initial_state(State),
    plan(State, Plan, _).
```

26(fruit colour)

```
% Define facts about fruits and their colors
fruit_color(apple, red).
fruit_color(banana, yellow).
fruit_color(grape, purple).
fruit_color(orange, orange).
fruit_color(lemon, yellow).
fruit_color(cherry, red).
fruit_color(kiwi, green).
fruit_color(plum, purple).
```

```
fruit_color(peach, pink).
fruit_color(pineapple, brown).
% Rule to find the color of a fruit
find_fruit_color(Fruit, Color) :-
    fruit_color(Fruit, Color).
```

27.bfs

```
% Define edges of the graph with their costs
edge(a, b, 1).
edge(a, c, 3).
edge(b, d, 3).
edge(b, e, 6).
edge(c, e, 2).
edge(d, f, 1).
edge(e, f, 2).

% Define the heuristic values (estimated cost to reach the goal)
heuristic(a, 6).
heuristic(b, 4).
heuristic(c, 5).
heuristic(d, 2).
heuristic(e, 1).
heuristic(f, 0). % Goal node

% Best First Search algorithm
best_first_search(Start, Goal, Path) :-
    heuristic(Start, H),
    bfs([[Start, H]], Goal, [], Path).

% Helper predicate to implement BFS
bfs([[Goal|Path]|_], Goal, _, [Goal|Path]).
```

```
bfs([[Current|Path]|Rest], Goal, Visited, FinalPath) :-
    findall([Next, H, Current|Path],
            (edge(Current, Next, _),
             \+ member(Next, Visited),
             heuristic(Next, H)),
            Neighbors),
    append(Rest, Neighbors, NewFrontier),
    sort(2, @=<, NewFrontier, SortedFrontier),
    bfs(SortedFrontier, Goal, [Current|Visited], FinalPath).

% Query to find the path
find_path(Start, Goal, Path) :-
    best_first_search(Start, Goal, RevPath),
    reverse(RevPath, Path).
```

28(medical diagnosis)

```
% Define symptoms
symptom(john, fever).
symptom(john, cough).
symptom(john, headache).
symptom(mary, sore_throat).
symptom(mary, cough).
symptom(mary, fatigue).
symptom(tom, rash).
symptom(tom, fever).
symptom(tom, headache).

% Define diseases and their associated symptoms
disease(flu, [fever, cough, headache, fatigue]).
disease(cold, [cough, sore_throat, fatigue]).
disease(measles, [rash, fever, headache]).
```

```
% Rule to diagnose a disease based on symptoms
diagnose(Patient, Disease) :-
   symptom(Patient, Symptom1),
   symptom(Patient, Symptom2),
   symptom(Patient, Symptom3),
   disease(Disease, Symptoms),
   member(Symptom1, Symptoms),
   member(Symptom2, Symptoms),
   member(Symptom3, Symptoms).
```

29.forward chaining

```
rainy(chennai).
rainy(coimbatore).
rainy(ooty).
cold(ooty).
snowy(X):-rainy(X),cold(X).
```

30.backward chaining

```
% Facts
fact(sunny).
fact(weekend).
fact(raining).
fact(weekday).

% Rules
rule(go_beach) :-
   fact(sunny),
   fact(weekend).

rule(watch_movie) :-
```

```
    fact(raining).

rule(stay_home) :-
    fact(sunny),
    fact(weekday).

% To deduce a fact
deduce(Fact) :-
    fact(Fact).

deduce(Fact) :-
    rule(Fact),
    \+ fact(Fact),
    assertz(fact(Fact)),
    write('Derived: '), write(Fact), nl.
```