**Water jug**

x = 4, y = 3, target = 2

*BFS*

from collections import deque

def solve\_bfs(x, y, target):

queue = deque([(0, 0, [])]) # (x, y, path)

visited = set()

while queue:

curr\_x, curr\_y, path = queue.popleft()

if curr\_x == target or curr\_y == target:

return path + [(curr\_x, curr\_y)]

if (curr\_x, curr\_y) in visited:

continue

visited.add((curr\_x, curr\_y))

operations = [

('Fill X', x, curr\_y),

('Fill Y', curr\_x, y),

('Empty X', 0, curr\_y),

('Empty Y', curr\_x, 0),

('Pour X to Y', max(0, curr\_x + curr\_y - y), min(y, curr\_x + curr\_y)),

('Pour Y to X', min(x, curr\_x + curr\_y), max(0, curr\_x + curr\_y - x))

]

for operation, next\_x, next\_y in operations:

queue.append((next\_x, next\_y, path + [(curr\_x, curr\_y, operation)]))

return None

bfs\_solution = solve\_bfs(x\_capacity, y\_capacity, target\_amount)

if bfs\_solution:

print("BFS solution:")

for state in bfs\_solution:

print(state)

*DFS*

def solve\_dfs(x, y, target, curr\_x, curr\_y, path, visited):

if curr\_x == target or curr\_y == target:

return path + [(curr\_x, curr\_y)]

if (curr\_x, curr\_y) in visited:

return None

visited.add((curr\_x, curr\_y))

operations = [

('Fill X', x, curr\_y),

('Fill Y', curr\_x, y),

('Empty X', 0, curr\_y),

('Empty Y', curr\_x, 0),

('Pour X to Y', max(0, curr\_x + curr\_y - y), min(y, curr\_x + curr\_y)),

('Pour Y to X', min(x, curr\_x + curr\_y), max(0, curr\_x + curr\_y - x))

]

for operation, next\_x, next\_y in operations:

result = solve\_dfs(x, y, target, next\_x, next\_y, path + [(curr\_x, curr\_y, operation)], visited)

if result is not None:

return result

return None

dfs\_solution = solve\_dfs(x\_capacity, y\_capacity, target\_amount, 0, 0, [], set())

if dfs\_solution:

print("\nDFS solution:")

for state in dfs\_solution:

print(state)

*Memoization*

def solve\_memoization(x, y, target, curr\_x, curr\_y, memo):

if curr\_x == target or curr\_y == target:

return [(curr\_x, curr\_y)]

if (curr\_x, curr\_y) in memo:

return None

memo.add((curr\_x, curr\_y))

operations = [

('Fill X', x, curr\_y),

('Fill Y', curr\_x, y),

('Empty X', 0, curr\_y),

('Empty Y', curr\_x, 0),

('Pour X to Y', max(0, curr\_x + curr\_y - y), min(y, curr\_x + curr\_y)),

('Pour Y to X', min(x, curr\_x + curr\_y), max(0, curr\_x + curr\_y - x))

]

for operation, next\_x, next\_y in operations:

result = solve\_memoization(x, y, target, next\_x, next\_y, memo)

if result is not None:

return [(curr\_x, curr\_y, operation)] + result

return None

memo\_solution = solve\_memoization(x\_capacity, y\_capacity, target\_amount, 0, 0, set())

if memo\_solution:

print("\nMemoization solution:")

for state in memo\_solution:

print(state)

**8 Puzzle**

from heapq import heappop, heappush

def solve\_8puzzle(initial\_state):

goal\_state = "12345678\_"

moves = {"U": -3, "D": 3, "L": -1, "R": 1}

open\_list, closed\_set = [], set()

calculate\_cost = lambda state: sum(abs(i%3 - state.index(str((i+1)%9))%3) + abs(i//3 - state.index(str((i+1)%9))//3) for i in range(8))

heappush(open\_list, (calculate\_cost(initial\_state), 0, initial\_state))

while open\_list:

\_, \_, current\_state = heappop(open\_list)

closed\_set.add(current\_state)

print\_state(current\_state)

if current\_state == goal\_state:

return

zero\_index = current\_state.index("\_")

for move, step in moves.items():

new\_index = zero\_index + step

if 0 <= new\_index < 9 and (move != "L" or zero\_index % 3 != 0) and (move != "R" or zero\_index % 3 != 2):

new\_state = list(current\_state)

new\_state[zero\_index], new\_state[new\_index] = new\_state[new\_index], new\_state[zero\_index]

new\_state = "".join(new\_state)

if new\_state not in closed\_set:

heappush(open\_list, (calculate\_cost(new\_state), 0, new\_state))

print("No solution found.")

def print\_state(state):

for i in range(0, 9, 3):

print(state[i:i+3])

print()

initial\_state = "1234\_6758"

solve\_8puzzle(initial\_state)

**Constraint Satisfaction Problem**

*\*\*\*IMP: pip install python-constraint\*\*\**

from constraint import Problem

def solve\_csp(variables, domains, constraints):

problem = Problem()

for variable in variables:

problem.addVariable(variable, domains[variable])

for constraint in constraints:

problem.addConstraint(constraint[1], constraint[0])

return problem.getSolutions()

variables = ['A', 'B', 'C']

domains = {

'A': [1, 2, 3],

'B': [1, 2],

'C': [1, 2, 3]

}

constraints = [

(('A', 'B'), lambda a, b: a != b),

(('B', 'C'), lambda b, c: b < c)

]

solutions = solve\_csp(variables, domains, constraints)

if solutions:

print("Solution found:")

for solution in solutions:

print(solution)

else:

print("No solution found.")

**Tic-Tac-Toe**

*ṣ\*\*\*IMP: pip install easyAI\*\*\**

from easyAI import TwoPlayerGame, AI\_Player, Human\_Player, Negamax

class TicTacToe(TwoPlayerGame):

def \_\_init\_\_(self, players):

self.players = players

self.board = [0] \* 9

self.current\_player = 1

def possible\_moves(self):

return [i + 1 for i, e in enumerate(self.board) if e == 0]

def make\_move(self, move):

self.board[move - 1] = self.current\_player

def unmake\_move(self, move):

self.board[move - 1] = 0

def lose(self):

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

return any(all(self.board[c - 1] == self.opponent\_index for c in line) for line in lines)

def is\_over(self):

return not self.possible\_moves() or self.lose()

def show(self):

print("\n".join(" ".join([".", "O", "X"][self.board[3 \* j + i]] for i in range(3)) for j in range(3)))

def scoring(self):

return -100 if self.lose() else 0

if \_\_name\_\_ == "\_\_main\_\_":

ai\_algo = Negamax(6)

TicTacToe([Human\_Player(), AI\_Player(ai\_algo)]).play()

**Bayesian Network**

import pandas as pd

from pgmpy.modwels import BayesianModel

from pgmpy.estimators import MaximumLikelihoodEstimator

from pgmpy.inference import VariableElimination

heartDisease = pd.read\_csv('heart\_disease\_data.csv', na\_values='?')

model = BayesianModel([('age', 'fbs'), ('fbs', 'target'), ('target', 'restecg'), ('target', 'thalach'), ('target', 'chol')])

model.fit(heartDisease, estimator=MaximumLikelihoodEstimator)

infer = VariableElimination(model)

query\_result = infer.query(variables=['target'], evidence={'age': 37})

print(query\_result)

**MDP**

*Value Iteration*

REWARD = -0.01

DISCOUNT = 0.5

MAX\_ERROR = 1e-3

NUM\_ACTIONS = 4

ACTIONS = [(1, 0), (0, -1), (-1, 0), (0, 1)]

NUM\_ROW = 3

NUM\_COL = 4

U = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]

def getU(U, r, c, action):

dr, dc = ACTIONS[action]

newR, newC = r + dr, c + dc

if newR < 0 or newC < 0 or newR >= NUM\_ROW or newC >= NUM\_COL or (newR == newC == 1):

return U[r][c]

else:

return U[newR][newC]

def calculateU(U, r, c, action):

u = REWARD

u += 0.1 \* DISCOUNT \* getU(U, r, c, (action - 1) % 4)

u += 0.8 \* DISCOUNT \* getU(U, r, c, action)

u += 0.1 \* DISCOUNT \* getU(U, r, c, (action + 1) % 4)

return u

def valueIteration(U):

print("The initial state is:")

for row in U:

print(" | ".join(f"{val:5}" for val in row))

print("\nDuring the value iteration:")

iteration = 0

while True:

nextU = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]

error = 0

for r in range(NUM\_ROW):

for c in range(NUM\_COL):

if (r <= 1 and c == 3) or (r == c == 1):

continue

nextU[r][c] = max([calculateU(U, r, c, action) for action in range(NUM\_ACTIONS)])

error = max(error, abs(nextU[r][c] - U[r][c]))

U = nextU

for row in U:

print(" | ".join(f"{val:5.2f}" for val in row))

print()

iteration += 1

if error < MAX\_ERROR or iteration > 10: # Stop condition modified for faster execution

break

return U

U = valueIteration(U)

print("\nThe optimal policy is:")

for row in U:

print(" | ".join(f"{val:5.2f}" for val in row))

*Policy Iteration*

import random

REWARD = -0.01

DISCOUNT = 0.99

MAX\_ERROR = 1e-3

NUM\_ACTIONS = 4

ACTIONS = [(1, 0), (0, -1), (-1, 0), (0, 1)]

NUM\_ROW = 3

NUM\_COL = 4

U = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]

policy = [[random.choice(["Down", "Left", "Up", "Right"]) for \_ in range(NUM\_COL)] for \_ in range(NUM\_ROW)]

def getU(r, c, action, U):

dr, dc = ACTIONS[action]

newR, newC = r + dr, c + dc

if newR < 0 or newC < 0 or newR >= NUM\_ROW or newC >= NUM\_COL or (newR == newC == 1):

return U[r][c]

else:

return U[newR][newC]

def calculateU(r, c, action, U):

u = REWARD

u += 0.1 \* DISCOUNT \* getU(r, c, (action - 1) % 4, U)

u += 0.8 \* DISCOUNT \* getU(r, c, action, U)

u += 0.1 \* DISCOUNT \* getU(r, c, (action + 1) % 4, U)

return u

def policyEvaluation(policy, U):

while True:

nextU = [[calculateU(r, c, policy[r][c], U) for c in range(NUM\_COL)] for r in range(NUM\_ROW)]

if max(abs(nextU[r][c] - U[r][c]) for r in range(NUM\_ROW) for c in range(NUM\_COL) if (r <= 1 and c != 3) and not (r == c == 1)) < MAX\_ERROR \* (1 - DISCOUNT) / DISCOUNT:

return nextU

U = nextU

def policyIteration(policy):

U = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]

iteration = 0

print("The initial random policy is:")

printPolicy(policy)

print("\nDuring the policy iteration:")

while True:

U = policyEvaluation(policy, U)

unchanged = True

iteration += 1

print("\nIteration:", iteration)

for r in range(NUM\_ROW):

for c in range(NUM\_COL):

if (r <= 1 and c == 3) or (r == c == 1):

continue

maxAction = max(range(NUM\_ACTIONS), key=lambda action: calculateU(r, c, action, U))

if maxAction != policy[r][c]:

policy[r][c] = maxAction

unchanged = False

printPolicy(policy)

if unchanged:

return policy

def printPolicy(policy):

action\_symbols = ["Down", "Left", "Up", "Right"]

for row in policy:

print("|", end=" ")

for action in row:

print(action\_symbols[action].ljust(5), end=" ")

print("|")

# Convert policy from symbols to indices

for i in range(NUM\_ROW):

for j in range(NUM\_COL):

policy[i][j] = ["Down", "Left", "Up", "Right"].index(policy[i][j])

policy = policyIteration(policy)

print("\nThe optimal policy is:")

printPolicy(policy)