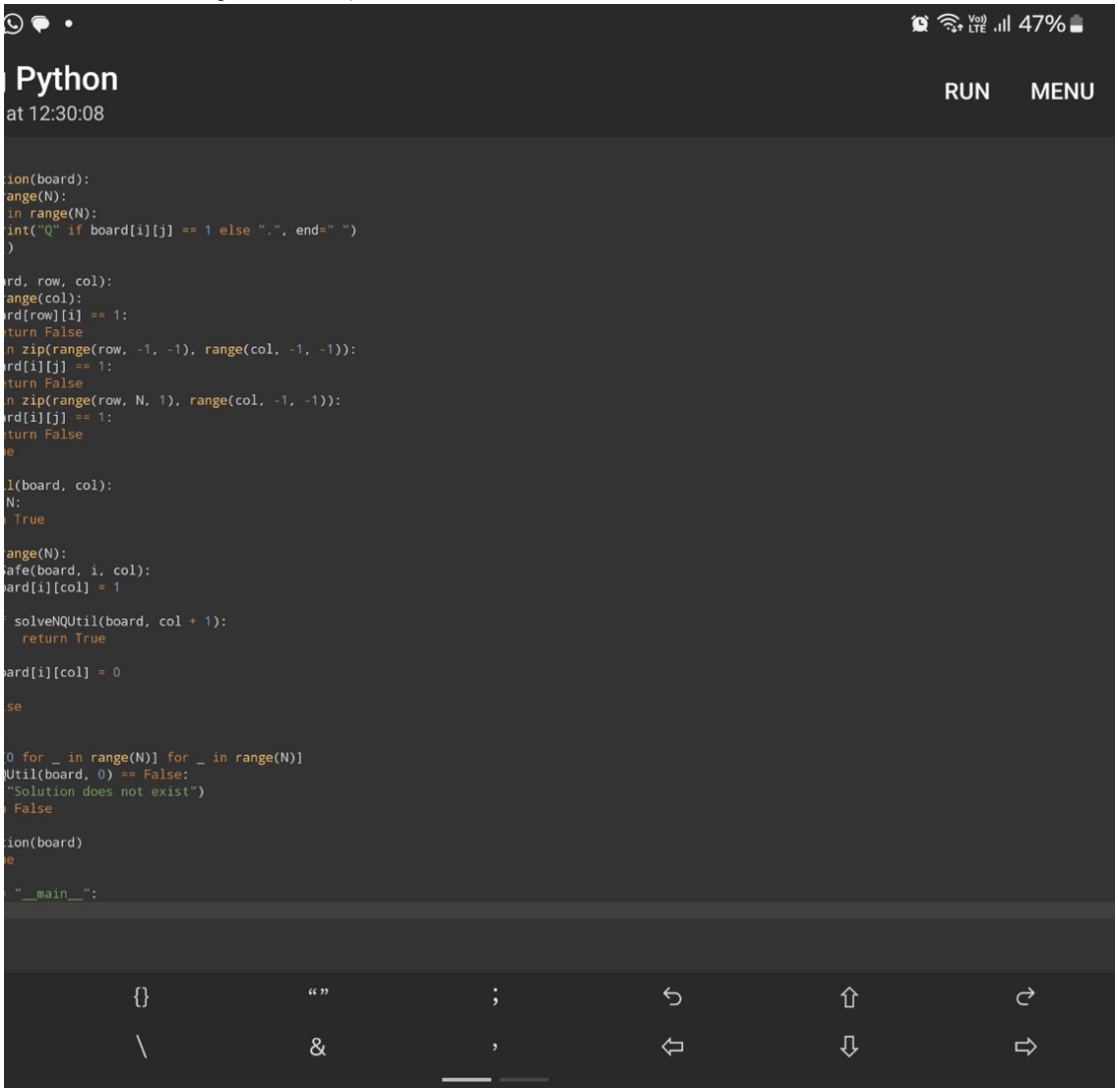
N-QUEENS, 2-a:


47%
Python
at 12:30:08
RUN
MENU
ion(board):
ange(N):
in range(N):
int("Q" if board[i][j] == 1 else ".", end=" ")
)
rd, row, col):
ange(col):
rd[row][i] == 1:
turn False
n zip(range(row, -1, -1), range(col, -1, -1)):
rd[i][j] == 1:
turn False
n zip(range(row, N, 1), range(col, -1, -1)):
rd[i][j] == 1:
turn False
e
l(board, col):
N:
True
ange(N):
afe(board, i, col):
ard[i][col] = 1
solveNQUtil(board, col + 1):
return True
ard[i][col] = 0
se
0 for _ in range(N)] for _ in range(N)]
Util(board, 0) == False:
"Solution does not exist")
False
ion(board)
e
"__main__":
{}
“”
;
\
&
,

1:32

Coding Python

Auto saved at 13:04:50

```python
class Node:
    def __init__(self, name, value=None, children=None):
        self.name = name
        self.value = value
        self.children = children if children is not None else []

def alpha_beta(node, depth, alpha, beta, max_player):
    if depth == 0 or node.value is not None:
        return node.value
    if max_player:
        v = float('-inf')
        for child in node.children:
            v = max(v, alpha_beta(child, depth-1, alpha, beta, False))
            alpha = max(alpha, v)
            if beta <= alpha:
                break
        return v
    else:
        v = float('inf')
        for child in node.children:
            v = min(v, alpha_beta(child, depth-1, alpha, beta, True))
            beta = min(beta, v)
            if beta <= alpha:
                break
        return v

# Create nodes and run alpha-beta pruning
A = Node('A', children=[Node('B', children=[Node('D', 3), Node('E', 12)]), Node('C', children=[Node('G', 9), Node('H', 2)])])
print(alpha_beta(A, 3, float('-inf'), float('inf'), True))
```

Tab {} “” ; = \ & ,

ALPHA-BETA implementation, 3a:

□

Python
at 12:16:12

RUN MENU

```
s np

1, p2):
linalg.norm(p1 - p2)

atrix(coordinate):
istance(p1, p2) for p2 in coordinate] for p1 in coordinate]

ution(matrix):
dom.sample(range(len(matrix)), len(matrix))

h(matrix, solution):
(matrix[solution[i-1]][solution[i]] for i in range(len(solution)))

ing(coordinate):
generate_matrix(coordinate)
lution = random_solution(matrix)
th = path_length(matrix, current_solution)

:
or = random.sample(range(len(matrix)), len(matrix))
or_path = path_length(matrix, neighbor)
ghbor_path < current_path:
rrent_solution = neighbor
rrent_path = neighbor_path

eak

rent_path, current_solution

p.array([[1,2], [30,21], [56,23], [8,18], [20,50], [3,4], [11,6], [6,7], [15,20], [10,9], [12,12]])
= hill_climbing(coordinate)
ution is \n", final_solution[1])
```

{} “” ; \ & ,

HILL_CLIMBING, 3-b:

□

12:15

Coding Python

Auto saved at 12:15:43

RUN

```python
import heapq

def a_star_search(grid, src, dest):
    open_list = [(0, src)]
    closed_list = set()
    parent_dict = {src: None}
    g_values = {src: 0}
    f_values = {src: calculate_h_value(*src, dest)}

    while open_list:
        _, current = heapq.heappop(open_list)
        if current == dest:
            print("The Path is ")
            path = []
            while current:
                path.append(current)
                current = parent_dict[current]
            path.reverse()
            for i in path:
                print("->", i, end=" ")
            print()
            return
        closed_list.add(current)
        for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            new_x, new_y = current[0] + dx, current[1] + dy
            if (0 <= new_x < len(grid)) and (0 <= new_y < len(grid[0])) and grid[new_x][new_y] == 1 and (new_x, new_y) not in closed_list:
                new_g = g_values[current] + 1
                new_f = new_g + calculate_h_value(new_x, new_y, dest)
                if (new_x, new_y) not in f_values or new_f < f_values[(new_x, new_y)]:
                    f_values[(new_x, new_y)] = new_f
                    g_values[(new_x, new_y)] = new_g
                    parent_dict[(new_x, new_y)] = current
                    heapq.heappush(open_list, (new_f, (new_x, new_y)))

    print("Failed to find the destination cell")

def calculate_h_value(x, y, dest):
    return ((x - dest[0]) ** 2 + (y - dest[1]) ** 2) ** 0.5

grid = [
    [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
    [1, 1, 1, 0, 1, 1, 1, 0, 1, 1],
    [1, 1, 1, 0, 1, 1, 0, 1, 0, 1],
    [0, 0, 1, 0, 1, 0, 0, 0, 0, 1],
    [1, 1, 1, 0, 1, 1, 1, 0, 1, 0],
    [1, 0, 1, 1, 1, 1, 0, 1, 0, 0],
    [1, 0, 0, 0, 0, 1, 0, 0, 0, 1],
    [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
    [1, 1, 1, 0, 0, 0, 1, 0, 0, 1]
]

src = [8, 0]
dest = [0, 0]
a_star_search(grid, tuple(src), tuple(dest))
```

Tab {} “” ; = \ & ,

A* Search, 4-a:

□

SOLVE WATER JUG ALGORITHM , 4-b:

□

Coding Python

Auto saved at 13:11:49

RUN M

```python
def find_best_move(board):
    """
    Finds the best move for a given game board.
    Args:
        board: A 2D list representing the game board.
    Returns:
        A tuple containing the row and column of the best move.
    """
    best_move = (0, 0)
    best_value = 0
    for row in range(len(board)):
        for col in range(len(board[0])):
            if board[row][col] > best_value:
                best_move = (row, col)
                best_value = board[row][col]
    return best_move

# Example usage:
board = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
]

best_move = find_best_move(board)
print("The Optimal Move is :")
print(f"ROW: {best_move[0]} COL: {best_move[1]}")
```

TIC-TACT-TOE algorithm ,5a:

Coding Python

Auto saved at 13:19:32

RUN MENU

```python
from heapq import heappush, heappop
import copy

n = 3  # Size of the puzzle (3x3)
moves = [(1, 0), (0, -1), (-1, 0), (0, 1)]  # Down, Left, Up, Right

class Node:
    def __init__(self, mat, empty_pos, level, cost, parent=None):
        self.mat = mat
        self.empty_pos = empty_pos
        self.level = level
        self.cost = cost
        self.parent = parent

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

def calculate_cost(mat, goal):
    return sum(mat[i][j] != goal[i][j] and mat[i][j] != 0
               for i in range(n) for j in range(n))

def create_node(mat, empty_pos, new_pos, level, parent, goal):
    new_mat = copy.deepcopy(mat)
    x1, y1 = empty_pos
    x2, y2 = new_pos
    new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2], new_mat[x1][y1]
    cost = calculate_cost(new_mat, goal) + level
    return Node(new_mat, new_pos, level, cost, parent)

def print_path(node):
    if node:
        print_path(node.parent)
        for row in node.mat:
            print(row)
        print()

def solve(initial, empty_pos, goal):
    pq = []
    root = Node(initial, empty_pos, 0, calculate_cost(initial, goal))
    heappush(pq, root)

    while pq:
        current = heappop(pq)
        if current.cost == current.level:
            print_path(current)
            return
        x, y = current.empty_pos
        for dx, dy in moves:
            nx, ny = x + dx, y + dy
            if 0 <= nx < n and 0 <= ny < n:
                child = create_node(current.mat, (x, y), (nx, ny),
                                    current.level + 1, current, goal)
                heappush(pq, child)

# Example usage
initial = [[1, 2, 3], [5, 6, 0], [7, 8, 4]]
goal = [[1, 2, 3], [5, 8, 6], [0, 7, 4]]
empty_pos = (1, 2)
solve(initial, empty_pos, goal)
```

6-a: Solve constraint satisfaction problem, 7-a:

10-a:

37%

g Python

ed at 13:22:50

RUN MENU

```
:
nit__(self, vertices):
f.V = vertices
f.graph = [[0] * vertices for _ in range(vertices)]

safe(self, v, color, c):
urn all(self.graph[v][i] == 0 or color[i] != c for i in range(self.V))

ve_util(self, m, color, v):
v == self.V:
 return True
 c in range(1, m + 1):
 if self.is_safe(v, color, c):
     color[v] = c
     if self.solve_util(m, color, v + 1):
         return True
     color[v] = 0
urn False
```

```
ve(self, m):
or = [0] * self.V
self.solve_util(m, color, 0):
 print("Solution exists: ", *color)
e:
 print("No solution")

sage
)
[0, 1, 1, 1], [1, 0, 1, 0], [1, 1, 0, 1], [1, 0, 1, 0]]
```

{} “” ; ⮌ ⇧ ⮎

\ & , ⇦ ⇩ ⇨

37%

Python

at 13:25:40

RUN MENU

```
_(self):
le = ['John', 'Mike', 'Tom']
emale = ['Jane', 'Lisa', 'Anna']
rent = {
hn': ['Mike', 'Anna'],
ne': ['Mike', 'Anna'],
ke': ['Tom'],
sa': ['Tom']


self, child):
next((p for p in self.male if child in self.parent.get(p, [])), None)

self, child):
next((p for p in self.female if child in self.parent.get(p, [])), None)

ther(self, child):
= self.father(child)
self.father(father) if father else None

ther(self, child):
= self.mother(child)
self.mother(mother) if mother else None

s(self, child, gender):
= [p for p in self.parent if child in self.parent[p]]
[s for p in parents for s in self.parent[p]
 if s != child and s in gender]

()
f Tom:", family.father('Tom'))
f Tom:", family.mother('Tom'))
her of Tom:", family.grandfather('Tom'))
her of Tom:", family.grandmother('Tom'))
 of Anna:", family.siblings('Anna', family.male))
of Mike:", family.siblings('Mike', family.female))
```

{} “” ; ⮌ ⇧ ⮎

\ & , ⇦ ⇩ ⇨