TJ Wiegman
ASM 591 AI
Lab 4
2024-10-02

Examples

Example 1: Map Coloring

Problem Description: Color a map such that no two adjacent regions have the same color. Use three colors: Red, Green, Blue.

Regions and Adjacencies:

```
Regions: A, B, C, D
Adjacent Pairs: (A,B), (A,C), (B,C), (B,D), (C,D)
```

In [1]:
```python
# Example 1: Map Coloring using Backtracking

def is_safe(region, color, assignment, adjacents):
    for neighbor in adjacents[region]:
        if neighbor in assignment and assignment[neighbor] == color:
            return False
    return True

def backtrack(assignment, regions, colors, adjacents):
    if len(assignment) == len(regions):
        return assignment
    unassigned = [r for r in regions if r not in assignment]
    region = unassigned[0]
    for color in colors:
        if is_safe(region, color, assignment, adjacents):
            assignment[region] = color
            result = backtrack(assignment, regions, colors, adjacents)
            if result:
                return result
            del assignment[region]
    return None

# Define regions and adjacents
regions = ['A', 'B', 'C', 'D']
adjacents = {
    'A': ['B', 'C'],
    'B': ['A', 'C', 'D'],
    'C': ['A', 'B', 'D'],
    'D': ['B', 'C']
}
colors = ['Red', 'Green', 'Blue']
```

```
solution = backtrack({}, regions, colors, adjacents)
print("Map Coloring Solution:", solution)
```

```
Map Coloring Solution: {'A': 'Red', 'B': 'Green', 'C': 'Blue', 'D': 'Red'}
```

Example 2: Sudoku Solver

Problem Description: Solve a 4x4 Sudoku puzzle where each row, column, and 2x2 subgrid must contain all numbers from 1 to 4.

In [2]:
```
# Example 2: Corrected Sudoku Solver with Backtracking

def is_valid(board, row, col, num):
    # Check if 'num' is not present in the current row and column
    for i in range(4):
        if board[row][i] == num:
            return False
        if board[i][col] == num:
            return False

    # Determine the starting indices of the 2x2 subgrid
    start_row, start_col = 2 * (row // 2), 2 * (col // 2)

    # Check if 'num' is not present in the 2x2 subgrid
    for i in range(start_row, start_row + 2):
        for j in range(start_col, start_col + 2):
            if board[i][j] == num:
                return False

    return True

def find_empty(board):
    for i in range(4):
        for j in range(4):
            if board[i][j] == 0:
                return (i, j)  # row, col
    return None

def solve_sudoku(board):
    empty = find_empty(board)
    if not empty:
        return board  # Puzzle solved
    row, col = empty

    for num in range(1, 5):
        if is_valid(board, row, col, num):
            board[row][col] = num  # Tentatively assign num

            result = solve_sudoku(board)
            if result:
                return result  # Solution found

            board[row][col] = 0  # Backtrack
```

```python
    return None  # Trigger backtracking

# Corrected Sudoku Puzzle (0 represents empty cells)
sudoku_board = [
    [0, 0, 2, 0],
    [0, 3, 0, 1],
    [1, 0, 0, 0],
    [0, 4, 0, 0]
]

solution = solve_sudoku(sudoku_board)

if solution:
    print("Sudoku Solution:")
    for row in solution:
        print(row)
else:
    print("No solution exists for the given Sudoku puzzle.")
```

```
Sudoku Solution:
[4, 1, 2, 3]
[2, 3, 4, 1]
[1, 2, 3, 4]
[3, 4, 1, 2]
```

Example 3: N-Queens Problem

Problem Description: Place N queens on an N×N chessboard so that no two queens threaten each other.

In [3]:

```python
# Example 3: N-Queens Problem using Backtracking

def is_safe(board, row, col, N):
    # Check this row on left side
    for i in range(col):
        if board[row][i] == 'Q':
            return False
    # Check upper diagonal on left side
    for i, j in zip(range(row-1,-1,-1), range(col-1,-1,-1)):
        if board[i][j] == 'Q':
            return False
    # Check lower diagonal on left side
    for i, j in zip(range(row+1,N), range(col-1,-1,-1)):
        if board[i][j] == 'Q':
            return False
    return True

def solve_nqueens(board, col, N, solutions):
    if col == N:
        solution = [''.join(row) for row in board]
        solutions.append(solution)
        return
    for i in range(N):
        if is_safe(board, i, col, N):
```

```python
            board[i][col] = 'Q'
            solve_nqueens(board, col + 1, N, solutions)
            board[i][col] = '.'

def nqueens(N):
    board = [['.' for _ in range(N)] for _ in range(N)]
    solutions = []
    solve_nqueens(board, 0, N, solutions)
    return solutions

# Solve 4-Queens
solutions = nqueens(4)
print(f"Total solutions for 4-Queens: {len(solutions)}")
for sol in solutions:
    for row in sol:
        print(row)
    print()
```

```
Total solutions for 4-Queens: 2
..Q.
Q...
...Q
.Q..

.Q..
...Q
Q...
..Q.
```

Example 4: Scheduling Classes

Problem Description: Assign time slots to classes ensuring no student has overlapping classes.

Classes and Students:

```
Classes: Math, Physics, Chemistry
Students:
    Student1: Math, Physics
    Student2: Physics, Chemistry
    Student3: Math, Chemistry
```

Available Time Slots: Morning, Afternoon

In [4]:
```python
# Example 4: Scheduling Classes using Forward Checking

def is_consistent(class_assignment, new_class, time, constraints, assignments):
    for student in constraints.get(new_class, []):
        for assigned_class, assigned_time in assignments.items():
            if (assigned_class in constraints and
                student in constraints[assigned_class]):
                if assigned_time == time:
                    return False
```

```
    return True

def forward_checking(classes, time_slots, constraints, assignments, index=0):
    if index == len(classes):
        return assignments
    current_class = classes[index]
    for time in time_slots:
        if is_consistent(
            assignments, current_class, time, constraints, assignments
        ):
            assignments[current_class] = time
            result = forward_checking(
                classes, time_slots, constraints, assignments, index + 1
            )
            if result:
                return result
            del assignments[current_class]
    return None

# Define classes, time slots, and constraints
classes = ['Math', 'Physics', 'Chemistry']
time_slots = ['Morning', 'Afternoon','Evening']
constraints = {
    'Math': ['Student1', 'Student3'],
    'Physics': ['Student1', 'Student2'],
    'Chemistry': ['Student2', 'Student3']
}

solution = forward_checking(classes, time_slots, constraints, {})
print("Class Schedule Solution:", solution)
```

```
Class Schedule Solution: {'Math': 'Morning', 'Physics': 'Afternoon', 'Chemistry':
'Evening'}
```

Example 5: Cryptarithmetic Puzzle

Problem Description: Solve the puzzle where each letter represents a unique digit: SEND+ MORE = MONEY

In [5]:
```
# Example 5: Cryptarithmetic Puzzle using Backtracking

import itertools

def solve_cryptarithmetic():
    letters = ('S', 'E', 'N', 'D', 'M', 'O', 'R', 'Y')
    for perm in itertools.permutations(range(10), len(letters)):
        s, e, n, d, m, o, r, y = perm
        if m == 0:
            continue  # M must be 1 since MONEY is a 5-digit number
        send = s*1000 + e*100 + n*10 + d
        more = m*1000 + o*100 + r*10 + e
        money = m*10000 + o*1000 + n*100 + e*10 + y
        if send + more == money:
            return {'S': s, 'E': e, 'N': n, 'D': d, 'M': m, 'O': o, 'R': r, 'Y': y
```

```python
    return None

solution = solve_cryptarithmetic()
print("Cryptarithmetic Solution:", solution)
```

```
Cryptarithmetic Solution: {'S': 9, 'E': 5, 'N': 6, 'D': 7, 'M': 1, 'O': 0, 'R': 8,
'Y': 2}
```

Exercise Problems

Exercise 1: Graph Coloring

Problem Description: Color a given map of 5 regions using four colors such that no two adjacent regions share the same color.

Regions and Adjacencies:

```
Regions: A, B, C, D, E
Adjacent Pairs: (A,B), (A,C), (B,C), (B,D), (C,D), (D,E)
```

In [2]:

```python
# Exercise 1: Graph Coloring with 4 Colors

def is_safe(region, color, assignment, adjacents):
    for reg in adjacents[region]:
        assigned = reg in assignment.keys()
        if assigned and (assignment[reg] == color):
            return False
    return True

def backtrack(assignment, regions, colors, adjacents):
    if len(assignment) == len(regions):
        return assignment
    region = None
    for r in regions:
        if r not in assignment:
            region = r
            break
    for color in colors:
        if is_safe(region, color, assignment, adjacents):
            assignment[region] = color
            result = backtrack(assignment, regions, colors, adjacents)
            if result:
                return result
            del assignment[region]
    return None

# Define regions and adjacents
regions = ['A', 'B', 'C', 'D', 'E']
adjacents = {
    'A': ['B', 'C'],
    'B': ['A', 'C', 'D'],
    'C': ['A', 'B', 'D'],
```

```
    'D': ['B', 'C', 'E'],
    'E': ['D']
}
colors = ['Red', 'Green', 'Blue', 'Yellow']

# Find and print the solution
solution = backtrack({}, regions, colors, adjacents)
print("Graph Coloring Solution:", solution)
```

```
Graph Coloring Solution: {'A': 'Red', 'B': 'Green', 'C': 'Blue', 'D': 'Red', 'E':
'Green'}
```

Exercise 2: Latin Square

Problem Description: Fill a 4x4 Latin square where each row and column contains the numbers 1 to 4 without repetition. Some cells are pre-filled.

Initial Grid

```
_ 2 _ 4
3 _ _ _
_ _ 2 _
1 _ _ 3
```

Instructions:

- Implement the is_valid function to ensure no duplicates in rows and columns.
- Implement the find_empty function to locate the next empty cell.
- Complete the solve_latin_square function to solve the puzzle.

In [9]:
```
# Exercise 2: Latin Square Solver using Backtracking

def is_valid(grid, row, col, num):
    # Check row
    if num in grid[row]: return False
    # Check col
    for r in grid:
        if r[col] == num: return False
    # If neither row nor col were invalid...
    return True

def find_empty(grid):
    for r in range(len(grid)):
        for c in range(len(grid[r])):
            if grid[r][c] == 0: return (r,c)
    return False

def solve_latin_square(grid):
    empty = find_empty(grid)
    if not empty:
        return grid
    row, col = empty
```

```python
    for num in range(1, 5):
        if is_valid(grid, row, col, num):
            grid[row][col] = num
            result = solve_latin_square(grid)
            if result:
                return result
            grid[row][col] = 0
    return ["Not solveable"]

# Define the initial grid (0 represents empty cells)
latin_square = [
    [0, 2, 0, 4],
    [3, 0, 0, 0],
    [0, 0, 2, 0],
    [1, 0, 0, 3]
]

# Find and print the solution
solution = solve_latin_square(latin_square)
print("Latin Square Solution:")
for row in solution:
    print(row)
```

```
Latin Square Solution:
Not solveable
```

Exercise 3: Timetable Scheduling

Problem Description: Assign time slots to courses ensuring no student is assigned overlapping courses.

Courses and Students:

```
Courses: Biology, History, Art, Computer Science
Students:
Alice: Biology, Computer Science
Bob: History, Art
Charlie: Biology, Art
Diana: Computer Science, History
```

Available Time Slots: Morning, Afternoon, Evening

Instructions:

```
Implement the is_consistent function to ensure no student has
overlapping courses.
Complete the schedule_courses function to assign time slots
effectively.
```

In [10]:
```python
# Exercise 3: Timetable Scheduling with Forward Checking

def is_consistent(course, time, assignments, constraints):
```

```
    same_time = []
    for crs in assignments:
        if assignments[crs] == time: same_time.append(crs)
    students = constraints[course]
    for crs in same_time:
        for stu in students:
            if stu in constraints[crs]: return False
    return True

def schedule_courses(courses, time_slots, constraints, assignments, index=0):
    if index == len(courses):
        return assignments
    current_course = courses[index]
    for time in time_slots:
        if is_consistent(current_course, time, assignments, constraints):
            assignments[current_course] = time
            result = schedule_courses(
                courses, time_slots, constraints, assignments, index + 1
            )
            if result:
                return result
            del assignments[current_course]
    return None

# Define courses, time slots, and constraints
courses = ['Biology', 'History', 'Art', 'Computer Science']
time_slots = ['Morning', 'Afternoon', 'Evening']
constraints = {
    'Biology': ['Alice', 'Charlie'],
    'History': ['Bob', 'Diana'],
    'Art': ['Bob', 'Charlie'],
    'Computer Science': ['Alice', 'Diana']
}

# Find and print the schedule
schedule = schedule_courses(courses, time_slots, constraints, {})
print("Timetable Schedule:", schedule)
```

```
Timetable Schedule: {'Biology': 'Morning', 'History': 'Morning', 'Art': 'Afternoon'
, 'Computer Science': 'Afternoon'}
```

Exercise 4: KenKen Puzzle

Problem Description: Solve a 3x3 KenKen puzzle where each row and column contains numbers 1 to 3 without repetition. Additionally, cells are grouped with a target operation.

Puzzle Layout:

```
Cells (0,0), (0,1): Sum to 4
Cells (0,2), (1,2): Product to 3
Cells (1,0), (2,0): Difference to 1
Cells (1,1), (2,1): Sum to 3
Cell (2,2): Must be 2
```

Instructions:

```
Implement the is_valid function to enforce row, column, and cage
constraints.
Implement the find_empty function to locate the next empty cell.
Complete the solve_kenken function to solve the puzzle.
```

In [28]:

```python
# Exercise 4: KenKen Puzzle Solver using Backtracking

def operation(op, num1, num2):
    if op == "+": return num1+num2
    if op == "-": return max(num1,num2)-min(num1,num2)
    if op == "*": return num1*num2
    else: return None

def is_valid(grid, row, col, num, cages):
    # Check row
    if num in grid[row]:
        # print(f"RowError: {num} is already in {grid[row]}")
        return False, 0
    # Check col
    for r in grid:
        if r[col] == num:
            # print(f"ColError: {num} is already in {[x[col] for x in grid]}")
            return False, 0

    # Check cages
    for cage in cages:
        op, out = cages[cage]
        if len(cage) == 1 and cage[0] == (row, col):
            if num != out:
                # print(f"CagError: {num} should be {out}")
                return False, out
        elif (row,col) in cage:
            num1 = grid[cage[0][0]][cage[0][1]]
            if num1 == -1: return True, 0
            num2 = grid[cage[1][0]][cage[1][1]]
            if num2 == 0: num2 = num
            if operation(op,num1,num2) != out:
                # print(f"CagError: {num1}{op}{num2} must be {out}")
                return False, -1

    # Finally, if no violations so far...
    return True, num

def find_empty(grid):
    for r in range(len(grid)):
        for c in range(len(grid[r])):
            if grid[r][c] == 0: return (r,c)
    return False

def find_lowPriority(grid):
    for r in range(len(grid)):
        for c in range(len(grid[r])):
            if grid[r][c] == -1: return (r,c)
```

```python
def solve_kenken(grid, cages):
    empty = find_empty(grid)
    if not empty:
        empty = find_lowPriority(grid)
    if not empty:
        return grid
    row, col = empty
    force = 0
    for num in range(1, 4):
        valid, suggested = is_valid(grid, row, col, num, cages)
        if suggested: force = suggested
        if valid:
            grid[row][col] = num
            # print(f"Assigned {num} to ({row},{col})!")
            # print(grid)
            result = solve_kenken(grid, cages)
            if result:
                return result

    if force:
        grid[row][col] = force # just try it, might fix later
        # print(f"Assigned {force} to ({row},{col})!")
        # print(grid)
        solve_kenken(grid,cages)
    return ["Not solveable"]

# Define the initial grid (0 represents empty cells)
kenken_grid = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]
]

# Define cages with their target and operation
cages = {
    ((0,0), (0,1)): ('+', 4),
    ((0,2), (1,2)): ('*', 3),
    ((1,0), (2,0)): ('-', 1),
    ((1,1), (2,1)): ('+', 3),
    ((2,2),): (None, 2)
}

# Find and print the solution
solution = solve_kenken(kenken_grid, cages)
print("KenKen Solution:")
for row in solution:
    print(row)
```

```
KenKen Solution:
Not solveable
```

Exercise 5: Word Ladder

Problem Description: Transform one word into another by changing one letter at a time, ensuring each intermediate word is valid. Find a sequence from "COLD" to "WARM".

Valid Words: COLD, CORD, CARD, WARD, WARM

Instructions:

```
Implement the is_adjacent function to determine if two words differ
by one letter.
Implement the word_ladder function using backtracking to find a
valid transformation sequence.
```

In [46]:

```python
# Exercise 5: Word Ladder Solver using Backtracking

def is_adjacent(word1, word2):
    diff = 0
    for i in range(len(word1)):
        if word1[i] != word2[i]: diff += 1
    if diff == 1: return True
    else: return False

def word_ladder(start, end, word_list, path={}):
    for word in word_list:
        if is_adjacent(start, word) and word not in path:
            path[start] = word
            if word == end:
                return list(path.keys()) + [end]
            else:
                return word_ladder(word,end,word_list,path)

# Define start, end, and word list
start_word = "COLD"
end_word = "WARM"
valid_words = ["COLD", "CORD", "CARD", "WARD", "WARM"]

# Find and print the word ladder
ladder = word_ladder(start_word, end_word, valid_words)
print("Word Ladder:")
print(ladder)
```

```
Word Ladder:
['COLD', 'CORD', 'CARD', 'WARD', 'WARM']
```