**Department of Computing**

**CS370: Artificial Intelligence**

**Class: BSCS-10AB**

**Lab 04: Constraint Satisfaction Problems**

**Date: 23-02-2023**

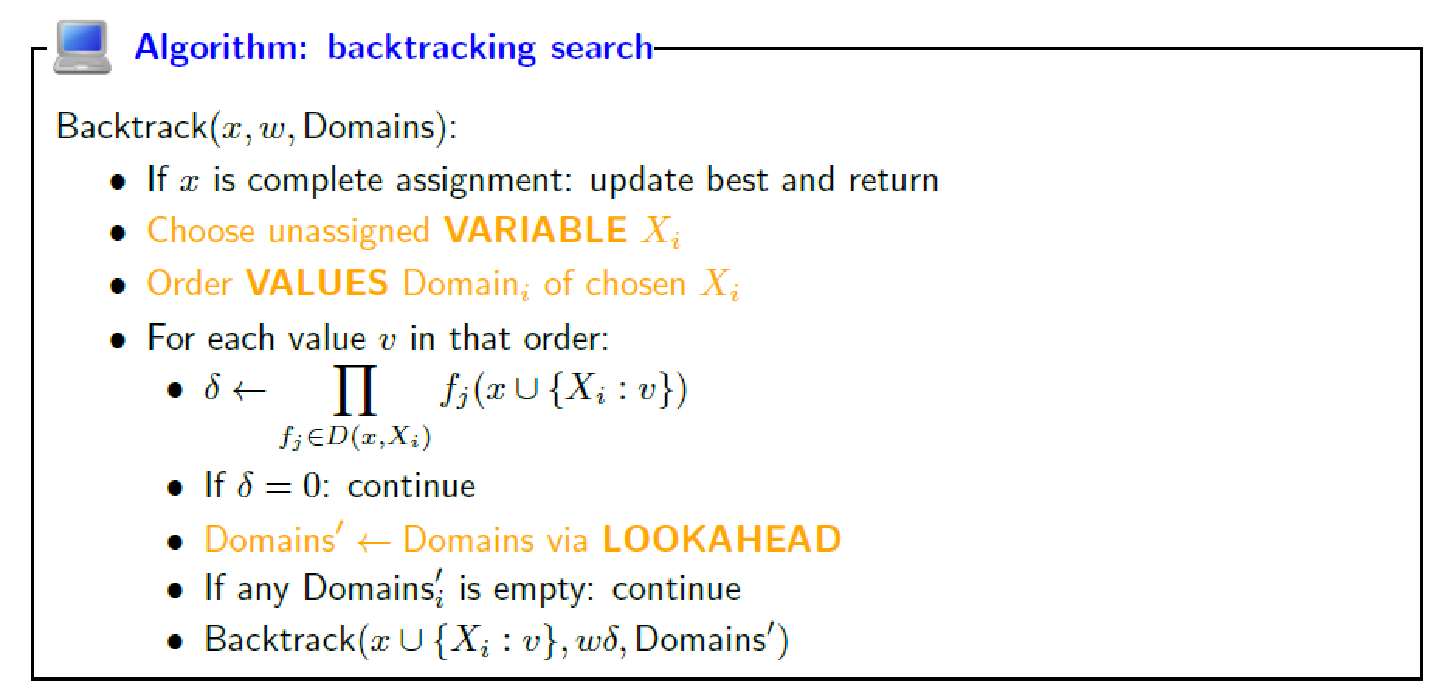
**Instructor: Dr. Hashir Kiani**

**Lab Engineer: Ms Shakeela Bibi**

# Lab 04: Constraint Satisfaction Problems

**Introduction**

Constraint Satisfaction Problems (CSPs) can be solved by using the backtracking search algorithm. Pseudocode for this algorithm is given below:



**Objective**

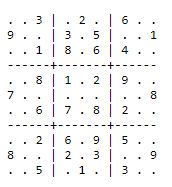
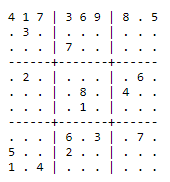
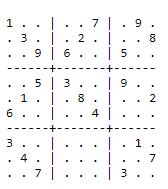
The objective of this lab is to implement the backtracking search algorithm for CSPs.

**Tools/Software Requirement**

Python, & its libraries

**Lab Task**

Write a python program that can solve a sudoku puzzle. Consider the sudoku puzzle as a constraint satisfaction problem and implement the backtracking search algorithm with arc consistency to solve the puzzle. Use the example puzzles below to test your implementation.

Use the following link to get solutions to any sudoku puzzle so you can check your result:

<https://anysudokusolver.com/>

**Description:**

* The main function of the program is solve\_sudoku, which takes in a partially filled-in Sudoku puzzle as a 9x9 2D array and attempts to fill in the empty cells to solve the puzzle.
* The solve\_sudoku function first finds an empty cell in the puzzle using the find\_empty\_location function and then iterates over the numbers 1-9 to try to fill in the cell with a valid number using the check\_location\_is\_safe function.
* If a number is found that can be placed in the cell, the function makes a tentative assignment and then recursively calls itself to try to solve the rest of the puzzle. If a solution is found, the process returns True. If no valid number can be placed in the cell, the function backtracks and tries a different number.
* If no solution can be found after trying all possible numbers, the function returns False.
* The print\_grid function is a utility function that prints the puzzle in a user-friendly way.
* The program also includes several helper functions (used\_in\_row, used\_in\_col, used\_in\_box, and check\_location\_is\_safe) that check whether a given number placement is valid based on the rules of Sudoku.

**Code:**

**# Define a function to find an empty location (0) in the Sudoku grid and return its row and column in a list "l"**

**def find\_empty\_location(arr, l):**

**for row in range(9):**

**for col in range(9):**

**if(arr[row][col] == 0):**

**l[0] = row**

**l[1] = col**

**return True**

**return False**

**# Define a function to check if a number "num" is already used in the given column "col"**

**def used\_in\_col(arr, col, num):**

**for i in range(9):**

**if(arr[i][col] == num):**

**return True**

**return False**

**# Define a function to check if a number "num" is already used in the given row "row"**

**def used\_in\_row(arr, row, num):**

**for i in range(9):**

**if(arr[row][i] == num):**

**return True**

**return False**

**# Define a function to print the Sudoku grid**

**def print\_grid(arr):**

**for i in range(9):**

**for j in range(9):**

**print(arr[i][j], end=" "),**

**print()**

**# Define a function to check if it is safe to place a number "num" in a given location (row, col) of the Sudoku grid**

**def check\_location\_is\_safe(arr, row, col, num):**

**# Check if the number is not already used in the same row, column, and box**

**return (not used\_in\_row(arr, row, num) and**

**(not used\_in\_col(arr, col, num) and**

**(not used\_in\_box(arr, row - row % 3,**

**col - col % 3, num))))**

**# Define a function to check if a number "num" is already used in the 3x3 box that contains the given cell (row, col)**

**def used\_in\_box(arr, row, col, num):**

**for i in range(3):**

**for j in range(3):**

**if(arr[i + row][j + col] == num):**

**return True**

**return False**

**# Define the AC3 algorithm**

**# Function to reduce domain of x based on arc consistency algorithm**

**def AC3(csp, queue=None):**

**def arc\_reduce(x, y):**

**removals = []**

**change = False**

**for vx in csp.domains[x].copy():**

**found = False**

**# Check if there exists a value in domain of y that satisfies the constraint with vx**

**for vy in csp.domains[y]:**

**if diff(vx, vy):**

**found = True**

**# If no such value is found, remove vx from domain of x**

**if(not found):**

**csp.domains[x].remove(vx)**

**removals.append((x, vx))**

**change = True**

**return change, removals**

**removals = []**

**# Initialize the queue with all arcs in the constraint graph**

**if queue is None:**

**queue = []**

**for x in csp.variables:**

**queue = queue + [(x, y) for y in csp.constraints[x]]**

**# Keep reducing domains until the constraint graph is arc consistent**

**while queue:**

**x, y = queue.pop()**

**b, r = arc\_reduce(x, y)**

**if r:**

**removals.extend(r)**

**if(b):**

**# If x's domain is empty, then no solution exists**

**if(len(csp.domains[x]) == 0):**

**return False, removals**

**# If we remove a value, we need to check all neighbors of x that are not y**

**else:**

**queue = queue + [(x, z) for z in csp.constraints[x] if z != y]**

**# If we reach here, the CSP is arc consistent**

**return True, removals**

**# Function to solve a Sudoku puzzle recursively**

**def solve\_sudoku(arr):**

**# Find the first empty cell in the puzzle**

**l = [0, 0]**

**if(not find\_empty\_location(arr, l)):**

**return True**

**row = l[0]**

**col = l[1]**

**# Try all numbers from 1 to 9 in the empty cell**

**for num in range(1, 10):**

**# Check if it is safe to place num in the cell**

**if(check\_location\_is\_safe(arr, row, col, num)):**

**# If it is safe, place num in the cell and continue recursively**

**arr[row][col] = num**

**if(solve\_sudoku(arr)):**

**return True**

**# If we reach here, the current placement of num did not lead to a solution, so backtrack**

**arr[row][col] = 0**

**# If we have tried all numbers from 1 to 9 and none led to a solution, then the puzzle is unsolvable**

**return False**

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

**grid = [[0 for x in range(9)]for y in range(9)]**

**grid = [**

**[3, 0, 0, 8, 0, 1, 0, 0, 2],**

**[2, 0, 1, 0, 3, 0, 6, 0, 4],**

**[0, 0, 0, 2, 0, 4, 0, 0, 0],**

**[8, 0, 9, 0, 0, 0, 1, 0, 6],**

**[0, 6, 0, 0, 0, 0, 0, 5, 0],**

**[7, 0, 2, 0, 0, 0, 4, 0, 9],**

**[0, 0, 0, 5, 0, 9, 0, 0, 0],**

**[9, 0, 4, 0, 8, 0, 7, 0, 5],**

**[6, 0, 0, 1, 0, 7, 0, 0, 3]]**

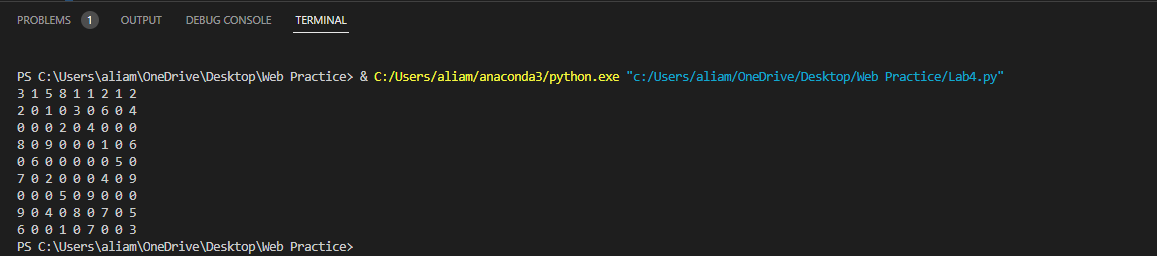
**if(solve\_sudoku(grid)):**

**print\_grid(grid)**

**else:**

**print("No solution exists")**

**Screenshot:**

****