Introduction to Artificial Intelligence

Lab 11 (Week 13) - Constraints Satisfaction Problems - Backtracking Search

2023 - 2024

May 12^{th}, 2024

Objectives

- Formulate the Sudoku game as a Constraint Satisfaction Problem (CSP).
- Implement a Backtracking strategy to solve the Sudoku game, using both recursive and iterative approaches.
- Implement the Arc Consistency strategy (AC3) to enhance the efficiency of the Backtracking algorithm by reducing the search space.

Overview:

In Lab 11, we will explore another method for formulating and solving problems known as Constraint Satisfaction Problems (CSPs). For more details on CSPs, please refer to CHAPTER 6 of the course material.

The main task of this lab is to implement a Sudoku solver using constraint satisfaction techniques. Sudoku is a logic-based puzzle game played on a 9x9 grid. The objective is to fill the grid with numbers from **1 to 9** such that each **row**, **column**, and **3x3 subgrid** contains all numbers **exactly once**. Some grid cells are initially filled with numbers; the number of such pre-filled grid cells is one of the factors that contribute to the difficulty level of the puzzle.

Your assignment involves developing a program capable of solving Sudoku puzzles by applying constraint satisfaction algorithms, specifically Backtracking Search. You will begin by formulating the Sudoku game as a CSP, specifying the following:

- **Set of Variables**: Represents the cells in the Sudoku grid.
- **Domains**: Specifies the possible values for each variable.
- **Constraints**: Defines the relations or conflicts between variables.

After formulating the problem, you will implement the backtracking algorithm using two different approaches: a recursive one and an iterative one. You will then compare these approaches in terms of complexity, considering execution time and memory usage (number of nodes explored).

Finally, you will enhance the basic backtracking algorithm by incorporating an Arc Consistency procedure, specifically AC3 with and without the Minimum Remaining Values (MRV) heuristic. This heuristic prioritizes variables that have the fewest remaining legal values (or domain values) available, aiming to reduce the branching factor and potentially improve the efficiency of the search algorithm.

You will conclude the lab with a comparison between the AC3-enhanced backtracking search algorithm and the simple backtracking search algorithm.

Your mission

You have been provided with the `Sudoku_CSP` class as an attached Python file.
Your mission is to perform the following tasks:

Task 1:

Given the following 9x9 grid (refer to Table 1), complete the remaining methods in the `Sudoku_CSP` class to properly formulate the Sudoku puzzle game as a CSP.

Table 1: Sudoko Initial Grid

5	3	0	0	7	0	0	0	0
6	0	0	1	9	5	0	0	0
0	9	8	0	0	0	0	6	0
8	0	0	0	6	0	0	0	3
4	0	0	8	0	3	0	0	1
7	0	0	0	2	0	0	0	6
0	6	0	0	0	0	2	8	0
0	0	0	4	1	9	0	0	5
0	0	0	0	8	0	0	7	9

Task 2:

The second task is divided into two parts:

- In the first part, use the pseudo-algorithm provided in 1 to implement a recursive version of the Backtracking Search algorithm to solve the Sudoku game.
- In the second part, you need to make the necessary adjustments to transform the recursive version into an iterative one.
- Execute both versions and compare their execution time, number of explored cells, and values.

Algorithm 1: Backtracking Search Algorithm

Input: *assignment* ;
Output: *Solution* or *None*;
if *assignment == length of variables* **then**
  return *assignment*;
end
var = Select-Unassigned-values(*assignment*);
for each *value* **in** Order-Domain-Values(*var*, *assignment*) **do**;
if *value is Consistent with assignment* **then**
  Add *value* to *assignment*;
  solution = Backtracking Search Algorithm(*assignment*);
  if *solution is not None* **then**
    return *solution*;
  end
  remove(*var*=*value*);
end
endFor;
return *solution*;

Task 3:

In this third task, your objective is to implement the AC3 algorithm described in 1. To achieve this goal, you are provided with three methods within the `Sudoku_CSP` class:

- `enforce_arc_consistency`: This is the main method responsible for enforcing arc consistency, utilizing other methods to achieve its goal.
- `initialize_arcs`: This method initializes a list of binary arcs ((i,j), (ii,jj)).
- `remove_inconsistent_values`: This method removes inconsistent values from a variable domain.
- `satisfies_constraints`: This method checks if two values satisfy or violate the constraints.

After implementing the AC3 algorithm, your final task is to utilize the Minimum Remaining Values (MRV) heuristic to select the next unassigned variable.

function AC-3(csp) **returns** false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components $(X,\ D,\ C)$
  local variables: $queue$, a queue of arcs, initially all the arcs in csp

  while $queue$ is not empty **do**
    $(X_i,\ X_j) \leftarrow$ REMOVE-FIRST($queue$)
    if REVISE(csp, $X_i,\ X_j$) **then**
      if size of $D_i\ =\ 0$ **then return** $false$
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add $(X_k,\ X_i)$ to $queue$
  return $true$

---

function REVISE(csp, $X_i,\ X_j$) **returns** true iff we revise the domain of X_i
  $revised \leftarrow false$
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      $revised \leftarrow true$
  return $revised$

Figure 6.3 The arc-consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be solved. The name "AC-3" was used by the algorithm's inventor (Mackworth, 1977) because it's the third version developed in the paper.

Figure 1: Arc Consistency: AC-3.