Vandan Vaishya

**CS 480 Spring 2024 Programming Assignment #02**

Due: **Sunday, March 31, 2024, 11:59 PM CST**

Points: **100**

**Instructions:**

1. Place **all your deliverables (as described below) into a single ZIP** file named:

LastName\_FirstName\_CS480\_Programming02.zip

1. Submit it to Blackboard Assignments section before the due date. **No late submissions will be accepted**.

**Objectives:**

1. (100 points) Implement and evaluate a constraint satisfaction problem algorithm.

**Problem description:**

Sudoku is a combinatorial, logic-based, number-placement puzzle. In classic Sudoku, the objective is to fill a 9 × 9 grid with digits so that each column, each row, and each of the nine 3 × 3 sub-grids that compose the grid contain all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a single solution (see Figure 1 below). [source: [Sudoku - Wikipedia](https://en.wikipedia.org/wiki/Sudoku)].

|  |  |
| --- | --- |
| 1. unsolved Sudoku puzzle | 1. solved Sudoku puzzle |
| A typical Sudoku puzzle, with nine rows and nine columns that intersect at square spaces. Some of the cells are filled with a number; others are blank cells to be solved. | The previous puzzle, showing its solution. |

*Figure 1: Sudoku puzzle: (a) unsolved, (b) solved* [source: [Sudoku - Wikipedia](https://en.wikipedia.org/wiki/Sudoku)]*.*

Your task is to implement in Python the following constraint satisfaction problem algorithms **(refer to lecture slides and/or your textbook for details and pseudocode)**:

* Brute force (exhaustive) search algorithm,
* Constraint Satisfaction Problem (CSP) back-tracking search,
* CSP with forward-checking and MRV heuristics,

and apply them to solve the puzzle (provided in a CSV file).

**NOTE: You ALWAYS have to start the search with AN EMPTY ASSIGNMENT. The fact that some cells (variables) are already filled with values DOES NOT mean that you start the search with those values already in. It MEANS that a domain for a corresponding variable is of size ONE (Dx = {ALREADY\_ASSIGNED\_VALUE}).**

Your program should:

* Accept two (2) command line arguments, so your code could be executed with

python cs480\_P02\_AXXXXXXXX.py MODE FILENAME

where:

* + cs480\_P02\_AXXXXXXXX.py is your python code file name,
  + MODE is mode in which your program should operate
    - 1 – brute force search,
    - 2 – Constraint Satisfaction Problem back-tracking search,
    - 3 – CSP with forward-checking and MRV heuristics,
    - 4 – test if the completed puzzle is correct.
  + FILENAME is the input CSV file name (unsolved or solved sudoku puzzle),

Example:

python cs480\_P02\_A11111111.py 2 testcase4.csv

If the number of arguments provided is NOT two (none, one, or more than two) or arguments are invalid (incorrect file, incorrect mode) your program should display the following error message:

ERROR: Not enough/too many/illegal input arguments.

and exit.

* Load and process input data file specified by the FILENAME argument (assume that input data file is ALWAYS in the same folder as your code - this is REQUIRED!).
* Run an algorithm specified by the MODE argument to solve the puzzle (or test if the solution is valid – MODE 4),
* Report results on screen in the following format:

Last Name, First Name, AXXXXXXXX solution:

Input file: FILENAME.CSV

Algorithm: ALGO\_NAME

Input puzzle:

X,6,X,2,X,4,X,5,X

4,7,X,X,6,X,X,8,3

X,X,5,X,7,X,1,X,X

9,X,X,1,X,3,X,X,2

X,1,2,X,X,X,3,4,X

6,X,X,7,X,9,X,X,8

X,X,6,X,8,X,7,X,X

1,4,X,X,9,X,X,2,5

X,8,X,3,X,5,X,9,X

Number of search tree nodes generated: AAAA

Search time: T1 seconds

Solved puzzle:

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

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

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

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

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

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

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

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

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

where:

* + AXXXXXXXX is your IIT A number,
  + FILENAME.CSV input file name,
  + ALGO\_NAME is the algorithm name (TEST for mode 4),
  + AAAA is the number of search tree nodes generated (0 for mode 4),
  + T1 is measured search time in seconds (0 for mode 4),
* Save the solved puzzle to INPUTFILENAME\_SOLUTION.csv file.
* In MODE 4 (test) your program should display the input puzzle along with a message

This is a valid, solved, Sudoku puzzle.

if the solution is correct and

ERROR: This is NOT a solved Sudoku puzzle.

if it is not correct.

**Input data file:**

Your input data file is a single CSV (comma separated values) file containing the Sudoku puzzle grid (see Programming Assignment #02 folder in Blackboard for sample files). The file structure is as follows:

X,6,X,2,X,4,X,5,X

4,7,X,X,6,X,X,8,3

X,X,5,X,7,X,1,X,X

9,X,X,1,X,3,X,X,2

X,1,2,X,X,X,3,4,X

6,X,X,7,X,9,X,X,8

X,X,6,X,8,X,7,X,X

1,4,X,X,9,X,X,2,5

X,8,X,3,X,5,X,9,X

You **CANNOT** modify nor rename input data files. Rows and columns in those files represent individual rows and columns of the puzzle grid as shown on Figure 1. You can assume that file structure is correct without checking it.

CSV file data is either:

* a character X corresponding unassigned (empty) grid cell,
* a positive integer (from the {1, 2, 3, 4, 5, 6, 7, 8, 9} set) corresponding to an assigned grid cell value.

**Deliverables:**

Your submission should include:

* Python code file(s). Your .py file should be named:

cs480\_P02\_AXXXXXXXX.py

where AXXXXXXXX is your IIT A number (this is REQUIRED!). If your solution uses multiple files, makes sure that the main (the one that will be run to solve the problem) is named that way and others include your IIT A number in their names as well.

* this document with your results and conclusions. You should rename it to:

LastName\_FirstName\_CS480\_Programming02.doc or pdf

Use testcase6.csv input data file and run all three algorithms to solve the puzzle. Repeat this search ten (10) times for each algorithm and calculate corresponding averages. Report your findings in the Table A below.

|  |
| --- |
| **Table A** |
| Algorithm | | Number of generated nodes | Average search time in seconds |
| Brute force search | | 102 | 4.750802755355835 |
| CSP back-tracking | | 391 | 0.004041823473843661 |
| CSP with forward-checking and MRV heuristics | | 81 | 0.0022949522191827946 |

What are your conclusions? Which algorithm performed better? What is the time complexity of each algorithm. Write a summary below

|  |
| --- |
| **Conclusions** |
| **Brute Force Search**  Nodes Generated: 102  Average Search Time: 4.7508 seconds  This approach attempts every possible number in each cell without using any strategies to reduce the search space. It's the least efficient in terms of time, significantly slower than the other methods. The large search time indicates that while the number of nodes generated isn't the highest, the process of generating and checking these nodes is computationally expensive.  Time Complexity: The worst-case time complexity is exponential, as it needs to check every combination of numbers in the grid. Specifically, it can approach O(9^N) for an N-cell Sudoku.  **CSP Backtracking**  Nodes Generated: 391  Average Search Time: 0.0040 seconds  CSP backtracking improves drastically on brute force by using the Sudoku rules to eliminate impossible numbers before trying them, reducing the search space. The number of nodes generated is higher than brute force, which indicates more checks but each check is significantly faster due to the reduced complexity of the decision at each step.  Time Complexity: This approach also has exponential time complexity in the worst case, but it's practically much faster than brute force due to early pruning of the search space. The exact complexity can vary, but it's generally better than brute force, approximating O(9^(N/2)) in bad cases due to early pruning.  **CSP with Forward-Checking and MRV Heuristics**  Nodes Generated: 81  Average Search Time: 0.0023 seconds  Incorporating forward-checking and Minimum Remaining Values (MRV) heuristics makes this the most efficient approach among those listed. It drastically reduces the number of nodes generated and, consequently, the time required to find a solution. The MRV heuristic helps in selecting the most constrained cell, potentially reducing the number of wrong paths taken early on, while forward-checking helps in pruning the search space dynamically as the algorithm progresses.  Time Complexity: Like other CSP methods, it's theoretically exponential in the worst case. However, the effective complexity is often much lower due to the significant pruning of the search space. The complexity is difficult to quantify due to the dynamic nature of the heuristics but is generally much better than simple backtracking.  **Conclusions**  Performance: The CSP with forward-checking and MRV heuristics outperformed both brute force and CSP backtracking in terms of speed, making it the most efficient algorithm among the three.  Efficiency: Despite generating the least number of nodes, the advanced CSP method completed searches much faster than the other approaches, showcasing the impact of intelligent search space pruning.  Preferred Method: For solving Sudoku puzzles, especially complex ones, CSP with forward-checking and MRV heuristics is the preferred method due to its superior balance of node generation and computational time efficiency. |