Matthew Muranaka

Objectives:

1. 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].

a) unsolved Sudoku puzzle

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

b) solved Sudoku puzzle

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

Figure 1: Sudoku puzzle: (a) unsolved, (b) solved [source: Sudoku - Wikipedia].

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).

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	1126584	28.09869234170001
CSP back-tracking	81	0.010631245700000047
CSP with forward-checking and MRV heuristics	81	0.011282174899999919

BackTrack				ForwardCheck		
Test	Nodes	Time		Test	Nodes	Time
1	1587	0.124467375		1	918	0.074904458
2	248	0.023889917		2	100	0.011834834
3	55733	4.205484375		3	94419	7.182856459
4	229	0.023016333		4	81	0.010141625
5	196	0.01963375		5	81	0.009855541
6	81	0.0087385		6	81	0.009127833
7	81	0.0086545		7	81	0.0095425

I also did a single test for each of the other test cases for both the BackTrack and ForwardCheck algorithms.

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

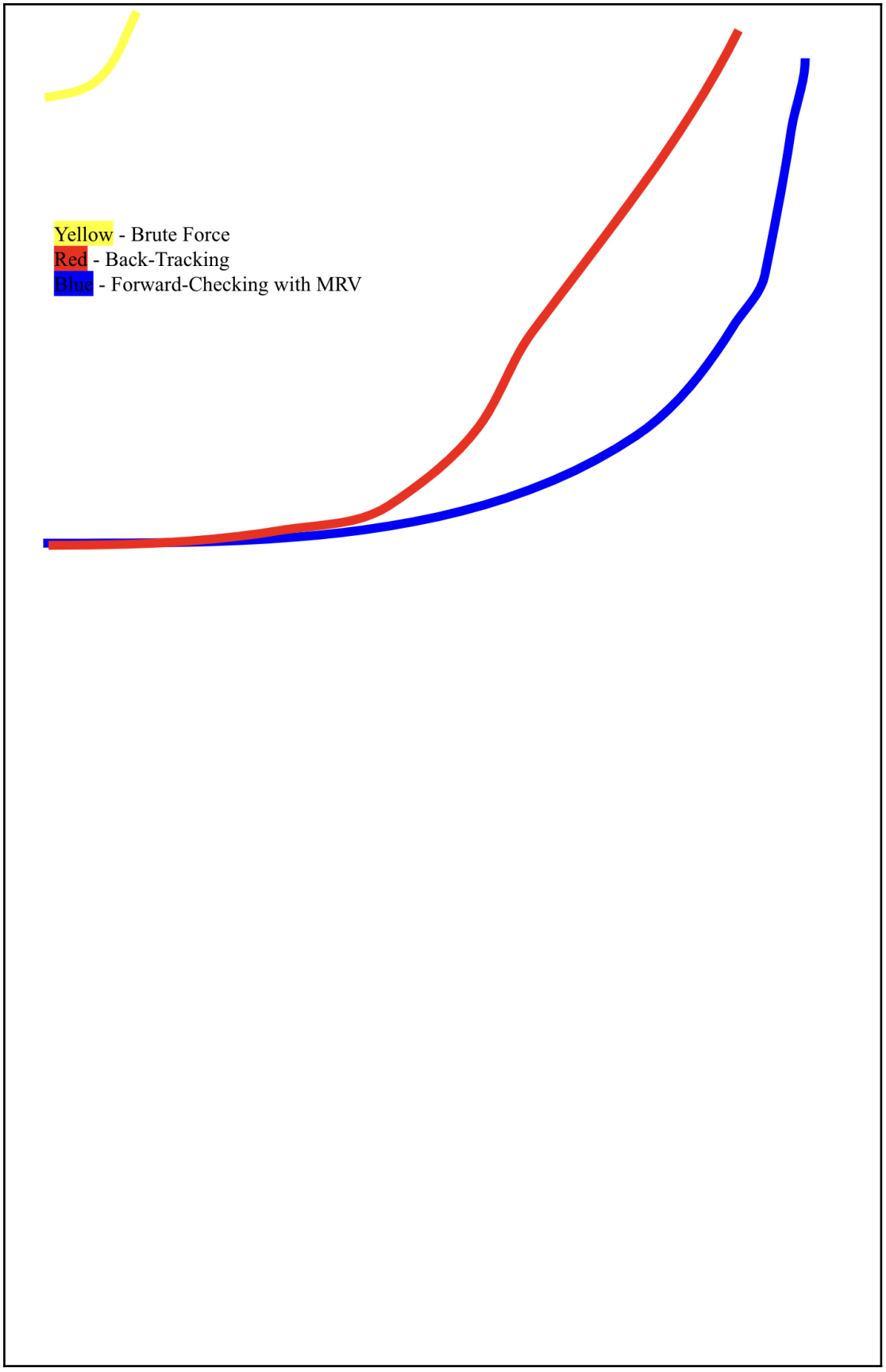
Conclusions

Please note that I created a new mode (5) which allows for the average testing needed above.

The averages for testcase6 show a slight advantage comparing the back-tracking and forward-checking algorithms when considering time. I believe this has to do with the extra time it takes in order to determine the MRV heuristic choices in the forward-checking algorithm, especially for a simple puzzle like this test case. Further, if you look at the other tests I conducted, the forward-checking algorithm produces significantly better time complexity while exploring less nodes.

One thing that is quite unexplainable is the increased time for testcase3, which is the test case with no solution. Not only does the forward-checking algorithm have a significantly longer time, the algorithm also expanded a considerable amount of search nodes more than the back-tracking algorithm.

I am not exactly sure how to evaluate time complexity, especially considering I don't know what the variable would be (maybe empty spaces), but I drew something of what the time complexity looks like as the amount of empty spaces in a Sudoku board increases.


Yellow - Brute Force
Red - Back-Tracking
Blue - Forward-Checking with MRV