CSE150
Program Assignment 3 Report

Yujia Li A98064697

yul200@ucsd.edu

Part 1: Description of the problem and the algorithms used in the problems

Overview:

In this project, we will develop algorithm to solve Futoshiki(sudoku) game. Although there are several different approaches to solving Futoshiki puzzle, you will be developing a generic binary constraint satisfaction problem(CSP)solver to solve the puzzle.

Discretion:

The Futoshiki Puzzle(sudoku) is played on a square grid, such as n by n matrix. The objective is to place the number 0 to n (integer), from n*1 to n*n (or whatever the dimensions are) such that each row and column contains each of the digits 0 to n. Some digits may be given at the start. In addition, inequality constraints are also initially specified between some of the squares, such that one must be higher or lower than its neighbor. And, These constraints must be honored as the grid is filled out.

Problem 1:

Implement the is_compete method that returns True when all variable in the CSP has been assigned.

Pseudocode:

a. Run through all the variables
b. Check every variable has assigned,
c. If False, return False , else return True

This is a checking condition method, first we run a loop of all the csp.variables and make a check condition inside of it variable.is_assigned(), if the variables has been assigned then return True, otherwise False. Basically, is_complete() method returns true when the csp assignment is complete, all variables in the csp has assigned.

Problem 2:

Implement the is_consistent method that returns True when the variable assignment to value is consistent. It does not violate any of the constraints associated with the given variable for the variables that have values assigned.

Pseudocode：

a. run a x in loop csp.constraints[variable] to get every constraint's variable
b. create a empty VAR
c. check if x first variable's name is same as variable's name are same
d. then assign the x's second to the VAR, otherwise assign x's first to the VAR
e. Check if variable has been assigned, if True:
    1) Assign value to variable
    2) Check x is satisfied, if True: return True, otherwise False
f. Return True for whole function

Basically, we need to implement this is_consistent method that will get the constraints variables, and checking variable is assigned in the list, if it is true then assign the value into variable. And check each single variable is satisfied by values assigned by pass in. if Yes, then Return True, otherwise, False.

Question 3:

Implement the basic backtracking algorithm in the backtrack() method. Check whenever searching next unassigned variable, check the pervious variable violate the constraint, such as A B C D, set A = red, B = blue and C = red, whenever searching to D, the C is violate with A.

Pseudocode：

a. Check csp is complete first, if true (base case), return assignment
b. Get an unassigned variable form csp
c. For loop run value in order_domain_values(csp,var)
d. Check condition of consistent(csp, var, value), if true
    1) csp.variables creates a backup of the current domain values
    2) sign the value to the direct of variable to value
    3) add var = value to assignment
    4) Check backtrack of csp condition, if true, return itself
e. Remove the any changes in the variable domains
f. Return None

The backtracking algorithm for constraint satisfy problems is varying functions of select_unassigned_variable() method and order_domian_values() method. First, we need implement is_complete() method that can check if all variables is complete for the base case, if it is return itself. And, get the next unassigned variable and assign to empty Var, if next unassigned variable is none then assignment is complete. Run a for loop of a list of domain's times with and each domain will pass into value. Check value is consistent with assignment and creates a backup of the current domain values. Then, assign the value into assignment index of var. Check the backtrack() method, if true return itself . At loop's end call rollback() method that will remove the value from variables.

Question 4:

Implement the AC3 algorithm, which depending on the arc parameter given, uses a queue to keep track of the arcs that need to be checked for inconsistency. Each xi, xj in turn is removed from the agenda and checked, if any values need to be deleted from the domain of xi, then every arc(xk, xj) pointing to xi must be reinserted on the queue for checking. There has two functions need to implement, revise() method and AC-3() method.

Pseudocode：

Function revise():

a. Run each xi in domain,
b. Run each xj in domain
    1) Check every variable satisfy the constraint between xi and xj
    2) Remove variable in xi domain

Function AC-3():

a. Check arcs is not none, then construct a deque with all the constraints
b. Run a deque is not empty:
    1) Pop every last element
    2) Call the revise to pass in the first element and second element
        a) Check if no element, return false
        b) run through the variable of constraints

c) add xk and xi into deque

Question 5:

This question is asking for improving the searching time of the algorithm by implementing heuristic functions in a better way. Both heuristic functions are select_unassigned_variable and order_domain_values, as mentioned before. Instead of using a native way to implement them, we are suggested to do some checking to speed up the return time.

Pseudocode：

Function select_unassigned_variable:

Here we are using the minimum-remaining-values(MRV) and degree heuristic, which means here are two rules:

First, always choose the variable with the smallest number of values left in its available domain;

Second, if there is a tie, choose the variable has largest number of constrains.

Pseudocode：

a) For every variable in csp.variables, check if it is assigned, if not
    1) check its domain size, if it smaller than the smallest domain size found so far, replace it and update
    2) if there is a tie, the variable has the same smallest domain size, check the constraints number of both, and if it has more constrains, replace and update

Function order_domain_values:

Here we use the least-constraining-value(LCV) heuristic function to reorder the variable list, sorted by the increasing order of the number choices for the neighboring variables in the constraint graph.

Pseudocode：

a. create a list to store all the variables with their potential number of choices for the neighbors. For every value of variable.domain:
    1) for every constrains of the current variable, count all its neighbor values that satisfied the constrain

2) store its value with its count into the list

b. sort the list with increasing count number and return

Question 6:

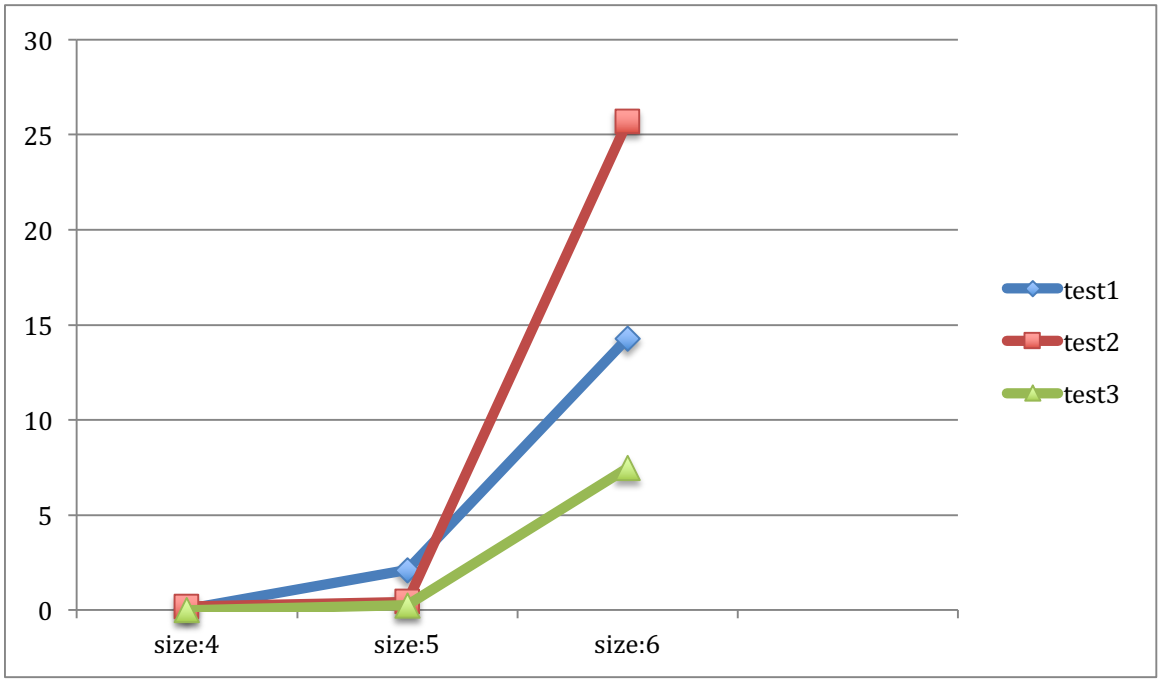
This problem is to group together all the algorithm and heuristic functions implemented from question 1 to question 5, so it can be used to solve real Futoshiki(sudoku) game. All the functions in problem 6 are:

a) inference, which called AC3 from question 4
b) backtracking_search, which is from question 3
c) is_complete, helper method for backtracking search, adapted from question 1
d) is_consistent, helper method from backtracking search, adapted from question 1
e) select_unassigned_variable, from question 5
f) order_domain_values, from question 6

Part 2: Analysis of solution rating

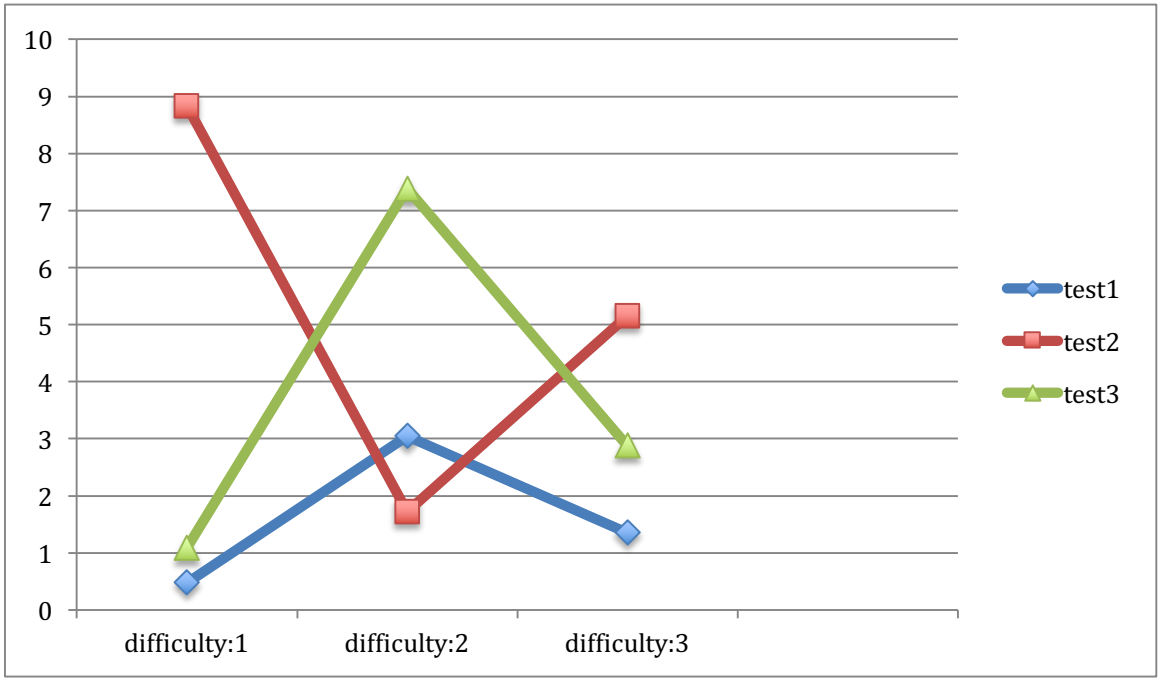
Index	Board Size	Puzzle No.	Difficulty	Time used
1	4	144	1	0.0668
2	5		1	2.1111
3	6		1	14.2796
4	4	190	1	0.1905
5	5		1	0.4419
6	6		1	25.6774
7	4	158	1	0.0063
8	5		1	0.2577
9	6		1	7.4727

Time solving puzzles with different board size



Index	Puzzle No.	Board size	Difficulty	Time used
1	160	5	1	0.4764
2			2	3.0486
3			3	1.3556
4	170	5	1	8.8281
5			2	1.7193
6			3	5.1488
7	163	5	1	1.0884
8			2	7.3834
9			3	2.8804

Time solving puzzles with different difficulty level



From the test data, it is not to find that as the board size increases, it takes longer time to solve a puzzle, and the incensement of the time is dramatic, the time takes to solve a 6 by 6 board is around 200 times the time takes to solve a 4 by 4 board. And based on our implementation, it takes too long to solve a puzzle with size bigger than 6 by 6.

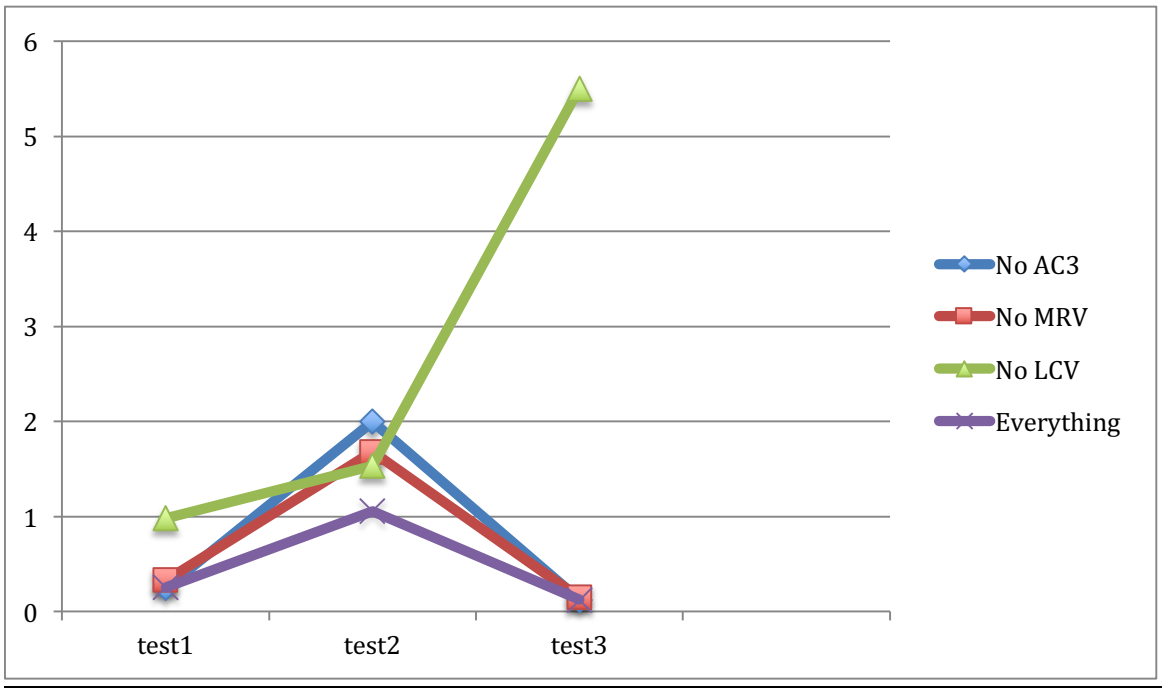
Also, as the difficulty of a board increase, it potentially takes longer to be solved, but it is not always certain. Sometimes a more difficult board can be solved faster than a easier board. So even a board is leveled to be harder, it is not necessary that it needs more time to be solved by searching algorithm we implemented.

Part 3: (Extra Point) Analysis of each type of heuristics

Index	Condition	Puzzle No.	Board size	Difficulty	Time used
1	No AC3	158	5	1	0.2493
2	No MRV	158	5	1	0.3312
3	No LCV	158	5	1	0.9835

4	Everything	158	5	1	0.2576
5	No AC3	162	5	1	1.9990
6	No MRV	158	5	1	1.6787
7	No LCV	158	5	1	1.5356
8	Everything	158	5	1	1.0583
9	No AC3	158	5	1	0.1243
10	No MRV	158	5	1	0.1489
11	No LCV	158	5	1	5.5049
12	Everything	158	5	1	0.1256

Time solving puzzles with different heuristic functions



Apparently, the reason we need heuristic functions and searching algorithm is to decrease the solving time of puzzle, and based on the data we collect, all of them are working towards minimize the searching time. Here AC3 stands for the back searching algorithm, MRV stands for minimum remaining values, and LCV stands for least-constraining value. Here in the graph, the purple line representing having all the algorithm and heuristic functions implemented always has the shortest

searching time.

Part 4: Contribution of each author

Yujia Li: By implementing problems 3 to 6, I have a deep understanding of backtracking searching algorithm, LCV and MRV heuristic functions. By writing lab report and analyze the data received by running tests, I have an idea about how different these different implementation affect the searching and its running time.