Xinyi Ma xim002@ucsd.edu
Yijun Zhang yiz160@ucsd.edu
Yuanchi Ha yuha@ucsd.edu

HW3 Writeup

Description:

P1_is_complete.py:

Problem: When all of the variables in the CSP have values assigned, return true, otherwise, return false.

Algorithm: Looped through all of variables in CSP(csp.variables). Then checked whether each variable has value, using var.is_assigned method; if any variable is not assigned, return false. Otherwise, return true.

P2_is_consistent.py:

Problem: When the variable assignment to value that are passed in parameter is consistent, return true, otherwise, return false.

Algorithm: Looped through all constraint requirements of the variable in parameter by passing it into csp.constraints[variable]. Then check whether its neighbor is assigned; if so, check whether the constraint is satisfied by calling constraint.is_satisfied function. If any constraint is not satisfied, return False, otherwise, return True.

P3_basic_backtracking.py:

Problem: Use the backtracking search to test whether there is a solution or not in backtrack(csp) method. Then the method will be called by backtracking_search(csp) to return the assignment if there is one.

Algorithm: The backtracking method is based on DFS, using recursion. We first checked whether the assignment is completed or not; if not, keep backtracking. At each recursive step, we first selected the next unassigned value, then check if there is a value in the domain that is consistent with the variable inside a for loop. If such a value exists, we assign it to the variable, and begin the next recursive step; otherwise, return false. This method incrementally builds up the solution, and terminate immediately each possibility that cannot possibly be a valid solution.

P4_ac3.py:

Problem: Implement ac3 method by checking the arcs between pairs of variables (x, y). It removes values in the domain of x, that aren't consistent with the constraint. When the domain of x has any values removed, all the arcs of constraints pointing to that pruned variable are added to the queues of arcs to be checked.

Algorithm: We keep a queue of arcs that needs to be checked. If it's empty, then return true. When it is not empty, we pop the newly added arc [x,y] out and check it: if x's domain is empty, then return false; otherwise, we remove the inconsistent value, and append new arcs related with x onto the queue. We also wrote a remove_inconsistent_values(csp, xi, xj) helper method. In this method, we loop though all the constraints between xi and xj, if any value in the domain of xi violates the constraints, we remove it.

P5_ordering.py:

Problem: Implement the variable heuristic using the minimum-remaining-values (MRV) and the degree heuristic as the tie-breaker in select-unassigned_variable(csp). Implement the value ordering, using the least-constraining-value (LCV) heuristic in order_domain_values(csp, variable).

Algorithm: 1. In select_unassigned_variable(csp) method, we first check if all variables have been assigned. If not, we loop through all the unassigned variables, and put them in a list, if there is only one element in the list, then we return it. If not, we sort them by the length of their domain to determine which has minimum remaining value. If the lengths of the last two variables' are not the same, then we return the one with the the minimum remaining value; otherwise, we put numbers of occurrences of each unassigned value into a list, and find out the index of the biggest number. Then return the unassigned variable with that value.

2. In order_domain_values(csp, variable) method, we first store the domain of the variable to a helper called copy, so that we can assign the domain back to original one afterwards. Then we assign all values in the domain to the variable, and append pairs of value and its corresponding number of violations to a list inside a for loop. After that, we store the domain of the variable back to variable.domain from copy. Then we sort the list by its number of violations, and return the list. In this algorithm, we also implemented a helper function to get the number of violations that would occur if variable is assigned to a certain value.

P6_solver.py:

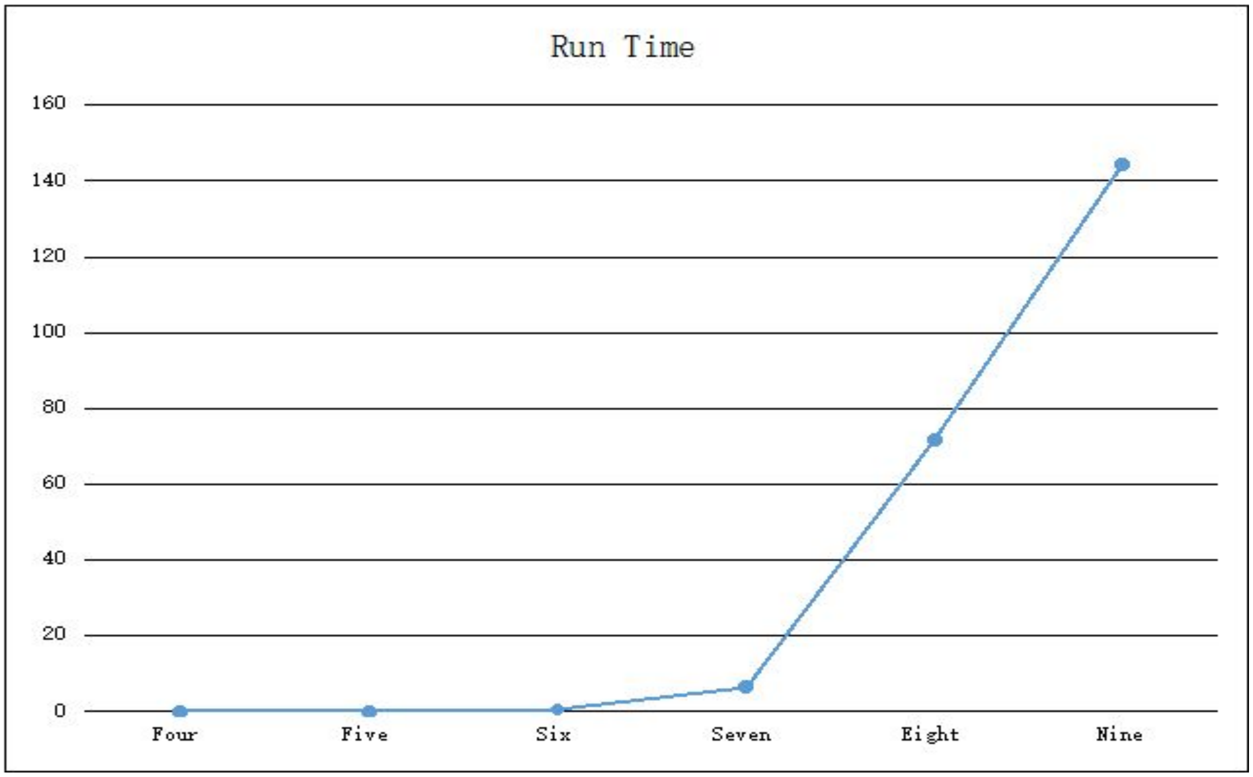
Problem: Complete a faster backtracking search algorithm by augmenting the basic backtracking algorithm, using the backtrack method from p3, with the MAC inference in p4, and the variable and value ordering heuristics in p5.

Algorithm: In order to use ac3, we implement it inside the inference funct. Then copied the ac3 and backtracking methods from previous problem into the script to be used. Likewise, to use variable and value ordering heuristics, we can implement it inside the inference func.

Run Time Analysis:

Varying board size:

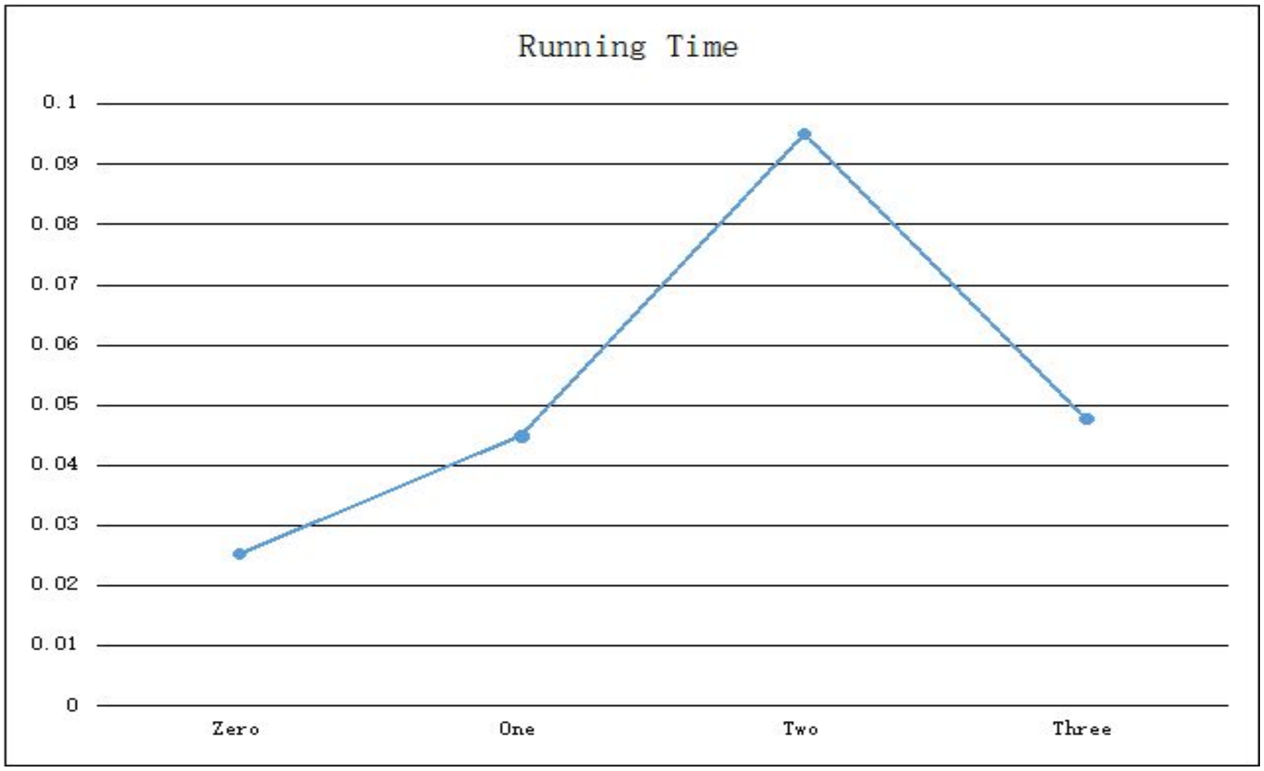
Difficulty	**Size**	**Time**
0	**4**	**0.00402593612671 seconds**
	5	**0.0218830108643 seconds**
	6	**0.554560899734 seconds**
	7	**6.45917201042 seconds**
	8	**71.6514010429 seconds**
	9	**144.435624123 seconds**



Varying Difficulty Level:

Size	**Difficulty**	**Time**
5	**0**	**0.0252919197083 seconds**
	1	**0.0448617935181 seconds**
	2	**0.0951108932495 seconds**
	3	**0.0477590560913 seconds**

When testing the relation between puzzle size and running time, we first set difficulty to 0 as a control variable, then tested puzzle of size between 4 to 9. As the result shown in graph, the time took running puzzle of size 4, 5 and 6 does not vary much, yet it increases substantially afterwards. We did the similar test with difficulty 1 puzzles, which took so long on our laptops to run the puzzle of level 8 and 9 that we had to manually interrupt. Thus, we concluded that the running time increases as the level of difficulty increases.



When testing the relation between puzzle difficulty and runtime, we first set size as 5 so that the different running times wouldn't be too short to be compared with each other. The interesting thing is while the running time of puzzle with difficulty level 1 is almost doubt of the puzzle with difficulty level 0, and the running time of puzzle with difficulty level 2 is almost double of the puzzle with difficulty level 1, the running time of puzzle with difficulty 3 dropped. So we did another test on puzzles with size 4, and got the similar result.

Extra Credit:

Size & difficulty	**Method**	**Time**
Size: 6 **Difficulty: 3** **ID: 124**	**P3/basic backtracking**	**17.626891374588013 seconds**
	Only Inference	**1.3534207344055176 seconds**
	Only Value & Variable ordering	**21.063018083572388 seconds**
	P6/all heuristics	**1.224064826965332 seconds**
Size: 5 **Difficulty: 3** **ID: 100**	**P3/basic backtracking**	**0.2656431198120117 seconds**
	Only inference	**0.2618448734283447 seconds**

	Only Value & Variable ordering	**0.07812619209289551 seconds**
	P6/all heuristics	**0.015626907348632812 seconds**
Size: 6 **Difficulty: 3** **ID: 89**	**P3/basic backtracking**	**7.5319859981536865 seconds**
	Only inference	**0.5781280994415283 seconds**
	Only Value & Variable ordering	**31.469210147857666 seconds**
	P6/all heuristics	**0.5781455039978027 seconds**

We tested the effect of different types of heuristics on a same puzzle. Our result shows that inference has great impact on the overall performance of the searching algorithm, which bring down the running time from 17 seconds to ~ 1 second on the first puzzle that we tested. Value and variable ordering helps in most cases, however the actual result depends on the randomness of the ordering of domains.

Summary:
Yuanchi Ha:
Helped to write p1, p2, and p3. In this assignment, I learned how to implement backtracking method, ac3, MRV, and LCV. During the process, I saw how much more concise these searching codes are compared to other searching codes and the different impacts of ac3, MRV, and LCV on backtracking.

Xinyi Ma:
Write p1, p2, p3 and p4. In this assignment, I learned that it is necessary to read and understand the given codes. I can use these codes directly instead of writing new functions. I also learned more about AC3 algorithm and understand the logic of it.

Yijun Zhang:
Wrote p5, p6 and helped debug p4. I learned how modules and classes work in Python, and saw how a smart optimization can tremendously help the actual running time of the searching algorithm.