Problem Set 1 Task 2

Name: Arno Dunstatter

eid: 1926976

# Strategy Overview

My strategy was to iteratively improve upon a recursive brute force backtracking method (called ‘naivebacktrack’ in my source code). After learning that this naive approach could not solve even problem A in a reasonable amount of time I restructured my code so that it was generalized, enabling me to test the naive approach on much simpler systems of equations and receive results in reasonable time. Thereafter, in backtrackV2 I added heuristics for choosing which variable to assign a value to in each recursive call, with these heuristics being embedded in my ‘variableSelector’ function. This version was able to solve problems A and B in reasonable time, but not C. In backtrackV3 I implemented forward checking via my ‘newDomains’ function which both reduced the scope of the search and synergized with my variable selection heuristics. This version was able to solve A, B, and C. I further attempted to implement a version 4 which ordered the values to assign, but this performed very poorly and I gave up on it due to time constraints.

The only libraries used were the copy library for its deepcopy() function and the importlib library for its reload() function.

# Pseudocode

For the sake of brevity, only my most successful versions will be explained here.

| goalTest(type):  partialPass is True fullPass is True for each constraint:  if the constraint is violated then partialPass and fullPass both are False and break  if the constraint is None then fullPass is False  if type is full then return fullPass  if type is partial then return partialPass  variableSelector():  if one variable has the smallest domain then return that variable if there’s a tie then from that tie return the variable with the most constraints if there’s a tie then from that tie return one arbitrarily  newDomains():  for each constraint:  if the constraint has only one unassigned variable:  for each value in the unassigned variable’s domain:  if the value violates the constraint remove it from the variable’s domain  backtrackV3():  if goalTest(full) then we found a solution so output all variables’ value assignments and exit  originalDomains = all variables’ domains  newDomains()  if any domain is empty then this is a dead end, return to last recursive call  variableToAssign = variableSelector()  originalValue = variableToAssign’s present value  for values in variableToAssign’s domain:  variableToAssign = val  nva = nva + 1  if goalTest(partial):  backtrackV3  reset all variables’ domains back to their originalDomains  variableToAssign = originalValue  nva = nva + 1 |
| --- |

# Pseudocode Explanation

The backtracking algorithm is based on the standard model of backtracking: testing for full goal completion at the very beginning so it knows if it can terminate before continuing into the next recursion. This algorithm also employs forward checking via the newDomains function which limits the search space as the algorithm enters into deeper recursive calls. Furthermore the algorithm employs heuristics to make inferences as to which variable to select for assignment next, prioritizing variables which have smaller domains and those whose value assignment will further limit the search space. The heuristic which prioritizes variables with small domains is further complemented by the forward checking employed by newDomains. Lastly this algorithm also utilizes a partial goal test to decide whether it's worthwhile to recurse or whether it makes sense to give up on an already obviously failed direction and look elsewhere for a solution.

# Discussion

The only mathematical pre-analysis which was employed was intuitive and then tested, rather than proven. Namely the 2nd heuristics of the variableSelector function seemed like a self-evident way of reducing the search space, and this was evidenced by the fact that it was the only real change made between the naiveBacktrack algorithm and the backtrackV2 algorithm, with the former being unable to compute a solution problem A over the course of a night of running, while the latter computed it in about a minute or so. As an aside: It should be noted here that while both heuristics were part of V2, since forward checking had not yet been implemented in V2 the domains never changed throughout V2’s execution, which is to say that for the given problems all their domains were of equal length for each recursion and only the second heuristic was relevant therein.

This implementation was designed to be generic in the sense that it should be able to solve any system of equations involving integers and elementary operations. I made my program generic by making only two assumptions about the problems which act as input: the variables’ values would be integers and the constraints are evaluable within python by default (i.e. they do not require any math libraries, trigonometric functions, etc). While debugging I have yet to find any valid problem which conforms to my aforementioned assumptions which are not solvable by my program.

One can directly test this program further with custom problems by either using the problem class directly, as shown on the next page, by providing to the problem class constructor a list of characters as the variables, a list of constraints as python executable strings, and a dictionary of dictionaries, or by using the rudimentary user interface described further in the ReadMe.txt.

Below are some examples of my program solving valid problems:

