Brendan Dwyer – 11717525

Data Structures and Algorithms – Zebra Problem Report

Overview of Solution:

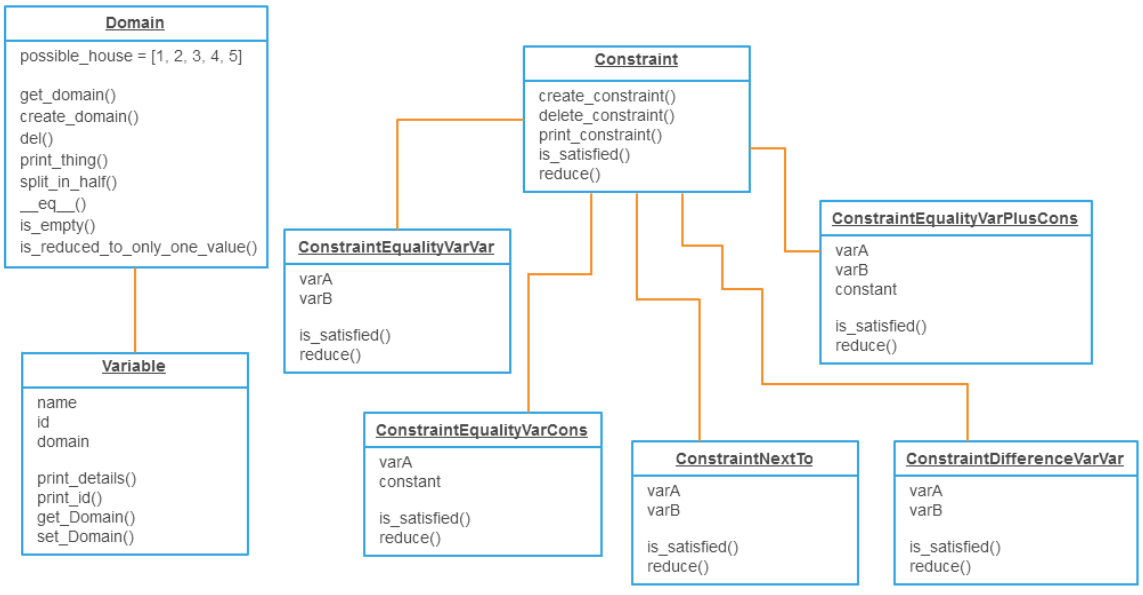
The puzzle is a list of fourteen clues which logically constrain the relationships between variables. The object of the puzzle is to reduce the domain of possibilities assigned to each variable from five potential versions to one remaining, certain value. From this we might deduce an answer to that most burning of questions: 'whose zebra is this and what is he drinking?'

This is achieved (or at least for the moment attempted) in this solution using an object oriented approach. Instances are created from the Variable Class, which inherits from the Domain Class and in turn confers the following list of initial possible values: [1, 2, 3, 4, 5]. Instance variables are then created from the various constraint classes, which all inherit from the abstract Constraint class (abstract because it has no __init__ method and other methods are all overridden in child-classes). There are 64 of these constraint instances; 50 of them enforce non-contradiction between related variables and the remaining 14 are the clues provided in the brief. Iterating over these constraints allows us to reduce the domain of possibilities for many of the variables.

After a certain point, progress can no longer be made by testing logical relationships known with certainty. Now, much like in sudoku or American elections, it is necessary to entertain a number of possible versions of the future (some perhaps terrifying) and check for logical contradictions in these versions given the constraints we have already established. Contradictions allow us to ascertain which domains can be further reduced. This approach can (hypothetically, not implemented yet) be effected using the deepcopy() method of the copy library and a tree data structure to handle the branching possibilities.

Structure of Solution:

UML diagram:



Brendan Dwyer – 11717525

Domain Reduction Logic:

Constraint

The Constraint class as outlined in the brief is an abstract class from which the more specific constraint classes inherit their methods. In this solution, the methods for Constraint are instantiated but implementation is left to the child classes, so the body for each method reads 'pass'. These are going to sound super dry, strap in.

ConstraintEqualityVarVar

This constraint is satisfied if there is any domain in common between the two variables being compared. I use a nested for loop in the 'is_satisfied' method to check every domain value of one variable against every domain value in its counterpart.

The reduction method of this constraint checks every domain value in variable A and checks if it is in the domain of variable B. If it is not (boolean NOT in the if statement) present in B, the value being checked is removed from the domain of A. The same process is repeated using a second set of nested 'for' loops to check domain values from variable B in the domain of A.

ConstraintEqualityVarCons

This constraint is satisfied if there is a value 'a' in the domain of varA such that a is equal to a constant. The 'is_satisfied' method returns a boolean statement which equates to True or False.

The reduction method of this constraint calls the 'set_Domain' method of varA using the constant as a parameter as we want to eliminate all domains other than the constant supplied. Note that the constant is enclosed in square brackets. Without them, any variable which satisfies ConstraintEqualityVarCons reduces the domain of that variable to an integer rather than a list containing one integer. This sounds obvious but it cost hours of my life wondering why the hell ConstraintNextTo was subsequently throwing errors exclusively on variables which had previously satisfied ConstraintEqualityVarCons. The brackets cast the value as a list.

ConstraintEqualityVarPlusCons

This constraint is satisfied by checking all values of varA's domain against all values of varB's to see if there is any value in varA's domain such that it is equal to one of varB's remaining domain values plus the constant.

It is reduced by removing values from varA's domain if the value of value 'a' minus the constant is not a value in varB's domain. In this implementation, a 'for' loop is used to check each value in varA against varB's domain. The same is repeated to remove values from varB's domain except when a value in varB's domain PLUS the constant (rearranging the equation) is not present in varA's domain.

ConstraintDifferenceVarVar

This constraint is not satisfied if the both domains are reduced to a single, identical value. In this solution, this was checked using a lengthy chained boolean expression and inverted using the 'NOT' statement, returning True or False.

It is reduced by checking both domains in turn to see if either of them are reduced to a single value. If so, index [0] returns that value, which is then removed from the other domain.

ConstraintNextTo

This constraint is satisfied if there are any values in the domain of varA plus or minus one which are also present in the domain of varB.

To reduce, a 'for' loop checks every value in the domain of varA to see if it has a neighbour in either direction of any value in the domain of varB. If there is no neighbour, that value is removed from the domain of varA. A second loop makes the same check in reverse for values in domain of varB to find neighbours in varB.

Domain Splitting

I haven't implemented this part of the problem yet. Constraints work and reductions work fine so the next step is to figure out how to split the domains of variables, test to see if a valid solution can be reached and iterate through all variables and constraints.

I plan to use the deepcopy() method to clone the values of objects rather than just creating a new pointer to the existing object. A tree data structure will handle the branching possibilities (and also is just much easier to visualise with the problem than a stack).

Various memorable problems:

'tuple object has no attribute Domain' in one of my Constraint classes. It turns out I didn't properly instantiate one single, lonely, troublesome Variable. Eventually fixed.

Originally used a few alternative methods for the various constraints, like appending to a list of common values in ConstraintEqualityVarVar. Decided this was a roundabout way of doing things, more consistent and readable to just remove unused values than to call the 'set_Domain' method with a list.