Zebra Puzzle Documentation.

Introduction

In this documentation I cover all the steps that where involved in my solution to the Zebra puzzle that we where assigned as a class project.

General implementation

For the most part of this code implementation of this project I followed the guidelines that where given in the assignment outline. The one major exception to this appears to be in the constraint class with the is_satisfied() function.

Here I have deviated and create a void function. Instead of returning true or False the function instead acts to ensure when it is run the constraint always become satisfied. In a number of cases as we carry out Domain splitting this can make the current puzzle layout invalid as it empties a domain. I explore this issue further in the relevant part of the documentation.

Overall I found the puzzle and enjoyable challenge. The most annoying part of me personally was spelling the work variable. I seem to have about three different ways of spelling the word at times in my code. This gives me the assurance that any one who where to steal my code would be easily identified due to the spelling mistakes!

Throughout my design of all of my code I have attempted to implement as general a solution to the problem as possible. This means that all of the inputs for the puzzle might change without having a need to rework the code extensively. This has also meant avoiding the use of constant throughout the puzzle. Whenever I fetch objects from storage arrays I attempt to do so in a way that would has no bearing on their location in the array.

Apart from this I found one of the more challenging elements of the puzzle and the area I believe caused the most people problems to be the handling object in data structures

Domain class

For the domain array there really is very little to comment on. This has been implemented according to spec of the assignment outline. The main purpose of this class was a storage structure for the possible domain that a variable could be in.

By default the init function creates the domain array and then create the domain values with 5 by default unless another specified amount is given.

The class has a number of functions that can act on the storage array and an equality function that is used to compared the domain array to another domain array when to domains are compared.

One exception to I made to the guidelines is the split_in_half() function. In my code this as marked as not finished. Due to architecture decisions I decided not to finish this class as I used a different method of to reach the solution.

In retrospect I would have finished the splitting function as I believe the architectecture of this path would have allowed for a more general program.

Variable class

The variable class of this program is used as a data container for each of the variables of the program. It contains the name of the variable, type of the variable and domains of the variable in the form of a domain class.

Constraint Class

The Constraint class is an abstract function that is inherited by each of the other constraint classes of the function. I have used validation to ensure a constraint can not be created that assigns a domain that is out of the domain range of the problem.

Each of the child classes implement a different constraint in roughly the same way. The major way I have changed the way I have implement the solution to the problem is by making is_satisfied() a void function. I had only realized this on reflection after completing the assignment.

In my each of the child classed the function is_satisfied() does not check if the constraint has been satisfied but instead work to implement the constrain on the variable set of the problem class. In the case of a collection of either variable domains or constraint that are logically incorrect this will invalidate the problem which will then be caught.

The constraint class Constraint_equality_var_plus_cons has a forth argument that allows the constraint to be used to set for domain just to the right of the first house of that the domain may be either side of the reference variable.

The constraint for now two variables of the same type being equal is set by a single constraint. This is assisted by the problem function the return_var_by_type() that will generate an array of all the different variable types.

In my constraint I have implemented

Problems Class

My problems class acts as a container for all the data structures and variables for the Zebra puzzle.

It works by first taking in a three dimensional array where the list list in the array will contain that variable types. Once the data is given in the correct format the __create() function will create all the variables of any size of problem. This would allow the puzzle to be expanded beyond the current scope to a much large problem.

There are four main attributes in this class __problem_solved = False, __variables_list = [], __constraint_set = [] and __multi_solution_list = []. Each of these either hold if the puzzle has been solved or acts as the respective data storage as is named.

Two other important attributes are variable_types and no of domains. Both of these are generated from the input array.

I have three function to print results. Print current results acts base function that allows the results of the current variables set in use to be printed. The other two functions act to check that a set of results has been found and then either prints out a single solution of a multi solution answer.

The function test if any domain empty acts as a validator for the current variable set. It allows us to determine if the current variables and their respective domains have the possibility of leading to a valid solution

set_variable_by_name allows a variables value to be set using the name of said variable and was mainly used for testing.

Return_variables_by_name is one of the more important functions that I used to make my problem more versatile. It allows me to return a pointer for the correct object by specifying the name type. This allows me to access my variable object held within the variable set arrays without having to know specifically where they are.

The last two functions test if problem solved y and count no of domains left are self explanatory by their name.

Reduction Function.

The reduction function acts to cycle through each of the constraints and apply this is_satisfied function to each of the variables.

For each constraint I first separate them by type. This is because each a different combination of input required to make the is_satisfied() function run correct.

For each constraint typed I then make input different information into the is_satisfied() function. Where the input required is a constant this can be put straight in by calling the constant attribute form the relevant constraint class.

In the case that one of the inputs is a domain another function must first be called. The function Return_variables_by_name() use the string to find the variable of the same same name and return it so that the is_satisfied() can take it as an input and act on it.

This function will continue to run until no more domains can be reduced. After this point it is time for the splitting function to finish the process and find the results

Splitting Function.

The splitter function uses a stack approach to solve the problem.

It works by first creating a number of data structures to store the solutions that might be found and possible solutions that are being worked on.

My solution to the problem I choose to avoid using a recursion method and instead opted for a stack method. I thought this might be easier to scale for a multicore processing environment and be easier to find multiple solutions to a problem.

The solution works by first taking in the current set of variables and their respective domain combinations. The algorithm then test that the current set is logically correct. This means that is does not already contain an empty domain for any variable.

Taking the first set of variables the search each variable till it finds a domain that that multiple values. This domain is then split according to the number of values that it has in.

For each value in the domain the following process is carried out.

A deepcopy is carried out on the variable objects array. The return from this is then assigned to a new variable array copy_of_variable_list. The program sets the domain for variable in variable equal to the [i] value of the original domain.

Then copy_of_variable_list is then set at the variable_set of the object. Reduction is carried out on the variable set.

The object now test if the problems has been solved. If it has the solution variable set is added to the solution array.

If the problem has not been solved the variable_set is tested to make sure no domain is empty. If every domain has at least one value the variable set is added to the list of possible solutions.

The function repeats this process with variable_set in the possible_variable_sets array until a solution is found or until the end of the arrays is reached.

Beyond the scope of the project

The code also works for scaled up versions of the project.

Puzzle solution with all constraints

The solution to the problem is
Variable English with type Nationality with current domains of [3]
Variable Spaniard with type Nationality with current domains of [4]
Variable Ukrainian with type Nationality with current domains of [2]
Variable Norwegian with type Nationality with current domains of [1]
Variable Japanese with type Nationality with current domains of [5]

Variable Red with type Color with current domains of [3]
Variable Green with type Color with current domains of [5]
Variable Ivory with type Color with current domains of [4]
Variable Yellow with type Color with current domains of [1]
Variable Blue with type Color with current domains of [2]

Variable Dog with type Pet with current domains of [4]
Variable Snails with type Pet with current domains of [3]
Variable Fox with type Pet with current domains of [1]
Variable Zebra with type Pet with current domains of [5]
Variable Horse with type Pet with current domains of [2]

Variable Snakes and Ladders with type Board Game with current domains of [3]
Variable Cluedo with type Board Game with current domains of [1]
Variable Pictionary with type Board Game with current domains of [2]
Variable Travel The World with type Board Game with current domains of [4]
Variable Backgammon with type Board Game with current domains of [5]

Variable Coffee with type Drink with current domains of [5]
Variable Milk with type Drink with current domains of [3]
Variable Orange Juice with type Drink with current domains of [4]
Variable Tea with type Drink with current domains of [2]
Variable Water with type Drink with current domains of [1]