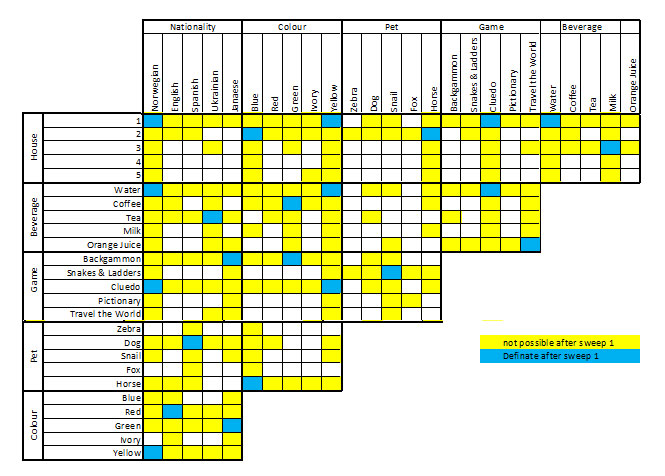
The Zebra Puzzle

The first thing I did to solve this puzzle was to do it on paper. I set up a document with all the possible variables and filled in the diagram below. I then went on to code how I solved this manually.



# House Domain

I then set up the domain class. This is the house domains class. It has the following functions:

* \_\_init\_\_: this sets up the list houses {1,2,3,4,5}
* domainReduction(house): This pops the “house” out of the domain. Returns true if any changes have been made to the domain list.
* printDomain: prints out the list of houses
* splitDomain: this returns two lists each containing half of the remaining possible domains. (While this function does work I can’t get the program to work with this function recursively. **Update: The program does work for the recursion but it takes a long, long, long time to run.** 6411 seconds.)
* remainDomains: Returns the list of domains
* setDomain(myList): Sets the domain list to the house list. Returns true if any changes have been made to the domain list.
* \_\_eq\_\_(otherDomain): compares this domain to another one and makes the two equal each other. Returns true if any changes have been made to the domain list.
* Is\_empty: returns true if the domain list is empty otherwise it returns false
* Is\_satisfied: returns true if the domain list contains only one item otherwise it returns false

# Variables

Next I set up the Variables class. This is a class representing each of the variable (zebra, dog, horse ect.) in the puzzle. This class implements the following functions:

* \_\_init\_\_(name, code): this this sets up the list houses {1,2,3,4,5}, name and code of each of the variable
* PrintVariable: this prints the all the details of the variable
* VariableDetails: This returns all the details of the variable

These function make direct calls to the relevant Domain functions and returns what they return

* GetDomains
* setDomains(myList)
* Is\_satisfied
* Is\_empty
* Split\_domain

# Constraints

The constrains file holds all the functions for the program to reduce the variable according to the constrains

The functions are as follows:

* Constraint\_equality\_var\_cons(Category, toChange, setHouse):

This function reduces the domains of the “toChange” variable to the “setHouse” house. It then removed the house from all the other domains in the category. This is used in the constraint “the Norwegian lives in the first house”. It sets the domain of the Norwegian variable to 1 and removes 1 from all the other variables in the nationality category.

This function returns True if any changes were made to the domains and false if the domains remain the same

* Constraint\_equality(var1, var2):

This function reduces the domains of each of the variables so they are the same. It is used in constraints like “the Englishman lives in the red house”. Only possible houses that are common to both variables will remain in either domain.

This function returns True if any changes were made to the domains and false if the domains remain the same

* Constraint\_toRight\_of(leftVar, rightVar):

This function checks if 0 is in the left var’s houses and if 1 is in the right var’s houses. If not it removes 0 from the left var and 1 from the right var. It continues right the way all all options. It is used for constraints like “The green house is immediately to the right of the ivory house.”

This function returns True if any changes were made to the domains and false if the domains remain the same

* Constraint\_NextTo(Var1, Var2):

This is very difficult to explain. If var1’s remaining domains are {2,4,5} this means that var2 can not be in house 2 as the houses on either side are not possibilities. To do this the function cycles through the possible option from 0 to 6. It checks to see if the houses on either side of this number is in either variables possible domains. If not it removes it from the other domain. It is used for constraints like “The man who plays Pictionary lives in the house next to the man with the fox.”

This function returns True if any changes were made to the domains and false if the domains remain the same

Sorry this is very difficult to explain.

* Constraint\_NoSet\_variable\_Match(varCategory):

This function checks if any of the variables in the category has been satisfied. If there is any it reduces all the other variables in the category so that they cannot have this domain. For example if Milk only has a domain of 5 then tea, coffee, orange Juice and water cannot be drunk in house 5. This is used as a check after all the other constraints has been run through.

This function returns True if any changes were made to the domains and false if the domains remain the same.

* Constraint\_Last\_Available(varCategory):

This function checks if any domain has only one possibility in the category. For example if the variables of beverage have the following domains. Milk {1,2,3,4,5}, tea {1,2,3,4}, coffee {1,2,3,4}, orange Juice {1,2,3,4}, and water {1,2,3,4}. The only beverage that it is possible for house 5 is milk. This function checks for this condition and sets milk to 5 in the above example.

This function returns True if any changes were made to the domains and false if the domains remain the same.

* Print\_all(allVars):

This function prints all of details for all of the variables in the allVars two dimensional array.

This function doesn’t return anything.

* Get\_available\_By\_house(varCategory):

This function is used to make things a bit prettier. It returns the name of the possible variables for each house. It returns it in a list format.

* Is\_all\_satisfied(allvars):

This function cycles though every option in the two dimensional array of all possible variables and check if each variable is satisfied.

It returns True if all the variables are satisfied and False if there are any variables with multiple options outstanding or any that are empty

* Is\_satisfied(elem):

This function is used to check if the element is satisfied.

It returns True if the variables are satisfied and False if there are multiple options outstanding or the variable is empty

* Is\_all\_empty(allvars):

This function cycles though every option in the two dimensional array of all possible variables and check if each variable is empty.

It returns True if any of the variables are empty otherwise it returns false.

* Print\_domainResults (allvars):

This function calls the get\_available\_By\_house for each category and formats these to print them out in a nice way.

It returns a list with all of the formatted options.

* Set\_up\_Variables ():

This function creates each of the variables. It then adds the 5 options to each category. I used a two dimensional array for this ease of use. The function returns the two dimensional array. They are in the format below:

* + AllVars
    - Nationality
      * Norwegian, English, Spanish, Ukrainian, Japanese
    - Beverages
      * Water, Coffee, Tea, Milk, OJ
    - Games
      * Backgammon, Snakes, Cluedo, Pictionary, Travel
    - Animals
      * Zebra, Dog, Snail, Fox, Horse
    - Colours
      * Blue, Red, Green, Ivory, Yellow
* Go\_Through\_Constrinats (allvars):

This function goes through each of the constraints given in the puzzle documentation. It then goes through and check if any other options are the last available and makes sure that no variables in any category equals a domain that is already set for that category. It then checks to see if any changes were made in the run through the constraints. It continues running through the constraints until no more changes can be made to any of the domains.

# Tree

The tree class sets up a simple tree and adds data to it. It has the following functions

* \_\_init\_\_(data):

This function initializes the tree and holds the data and child nodes of the tree. The data is the applicable allVars from the main program

* Add\_child(obj):

This function adds a child to the parent tree

* Get\_data:

This function returns the data in the node

# Zebra Puzzle Fast

This program sets up the variables and runs them through the constraints. As the puzzle cannot be solved with only the constraints given it then has some decisions to make. It sets up a tree with the variables as they stand as the data. It then calls process\_each passing in the tree that it has just set up. It then checks to see if there are any correct answers and prints these out to the screen.

The tree will have several copies of the same answer so the program checks for duplicate answers and prints out the correct ones on the screen.

Process\_each gets the data (variables) from the parent tree. It then looks at each variable in turn. If the variable is not satisfied it gets the possible remaining domains and runs the constraints again with the variable set to one of the possible remaining domains. As this is very difficult to explain the diagram below shows the logic of the function.

The answer that I got for the puzzle is:

house: 1 : Norwegian | Water | Cluedo | Fox | Yellow

house: 2 : Ukrainian | Tea | Pictionary | Horse | Blue

house: 3 : English | Milk | Snakes & Ladders | Snail | Red

house: 4 : Spanish | Orange Juice | Travel the World | Dog | Ivory

house: 5 : Japanese | Coffee | Backgammon | Zebra | Green

# Zebra Puzzle Slow

This program works in the same manner as the zebra Puzzle Fast program. This one though uses the domain splitting instead of setting each variable as it goes.. It does run and runs correctly and gives out the correct answer however it runs very very slowly. It takes about an hour to run. I can’t seem to figure out how to make this go any quicker.

Process Each(Parent Node)

Get the data (allvars) from the tree.

For each variable

(Norwegian, Spanish…..)

Deep copy parent

Is the variable satisfied

Get all the possible domains for this variable

For each of the possible domains

Set the variable = a possible domain

Run through the constraints

Create a tree with the data set to the new variable states

Is there any empty variables

Add the new tree as a child of the parent tree

Process each for the new tree

Yes

No

Yes

No