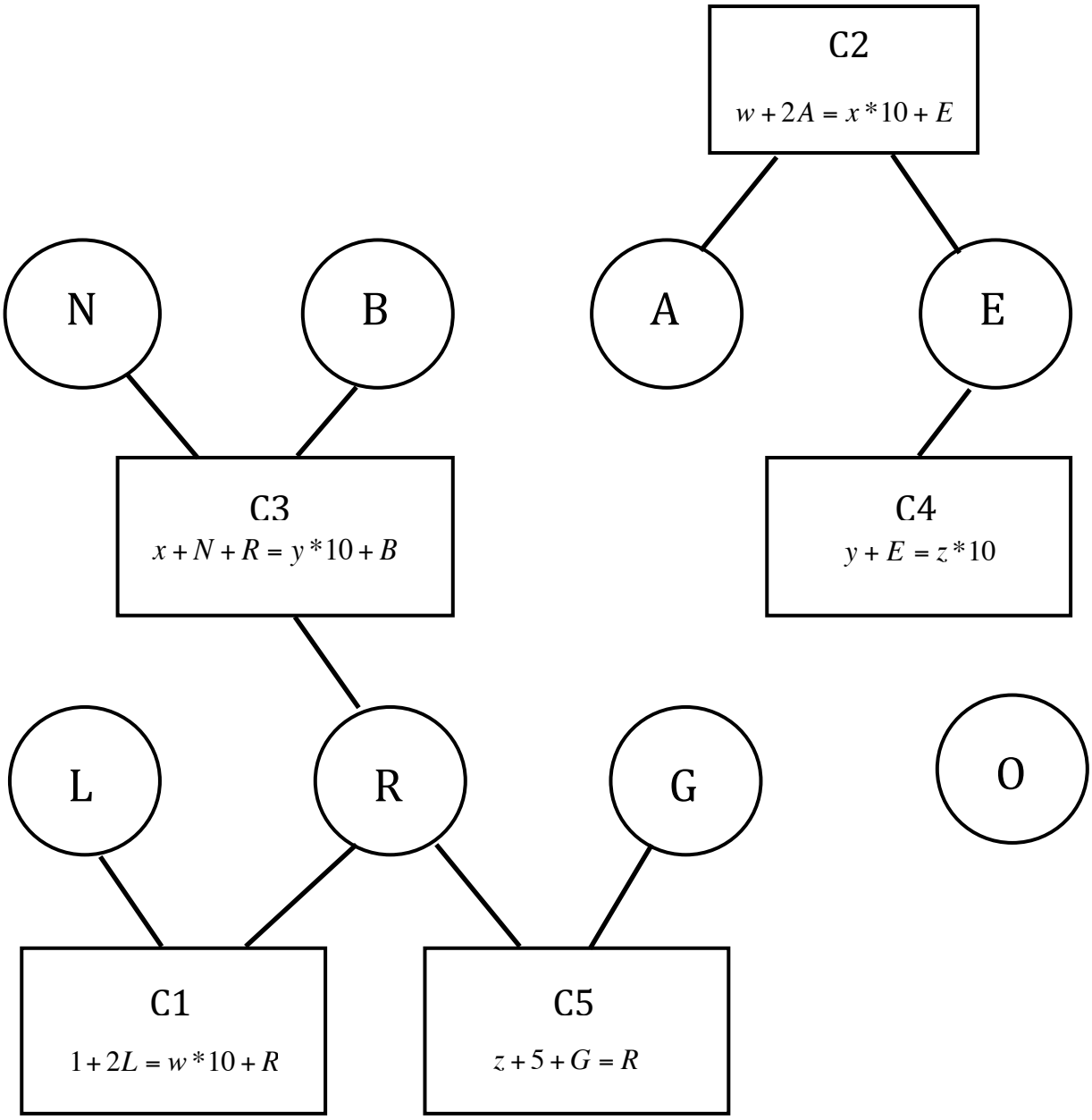
We're going to start this lab by finishing up our work on the domain and arc consistency approach to solving constraint satisfaction problems. We have been looking at this classic crypt-arithmetic problem:

> Find an assignment of the integers 0 – 9 to the letters in the words
>
> DONALD, GERALD and ROBERT such that:
>
> each integer is assigned to a unique letter;
>
> each letter is assigned to a unique integer;
>
> when the letters have been replaced by their
> assigned numbers, the following equation is satisfied:
>
> DONALD + GERALD = ROBERT.
>
> The value 5 is assigned to the letter D,
> and we figured out early on that 0 must be assigned to the letter T.

In *Lab 3* and *Lab 4* this problem was reformulated as follows:

> The reformulated problem has 12 variables, namely the letters A, B, E, G, L, N, O, and R, along with the nuisance parameters *w*, *x*, *y*, and *z*.
>
> A solution to the problem is an assignment of integers {1, 2, 3, 4, 6, 7, 8, 9 } to the variables A, B, E, G, L, N O, and R, and an assignment of integers { 0, 1, } to the nuisance parameters *w*, *x*, *y*, and *z*, satisfying these *symmetric inequality constraints*:
>
> $$\begin{array}{rrrrrr} A \neq B, & A \neq E, & A \neq G, & A \neq L, & A \neq N, & A \neq O, \quad A \neq R, \\ B \neq E, & & \cdots & & & B \neq R, \\ E \neq G & & \cdots & & & E \neq R, \\ \ddots & & \cdots & & & \vdots \\ & & & & N \neq O, & N \neq R, \\ & & & & & O \neq R; \end{array}$$
>
> and these *equation constraints*:
>
> $$1 + 2L = w * 10 + R, \qquad \text{(C1)}$$
>
> $$w + 2A = x * 10 + E, \qquad \text{(C2)}$$
>
> $$x + N + R = y * 10 + B, \qquad \text{(C3)}$$
>
> $$y + E = z * 10, \text{ and} \qquad \text{(C4)}$$
>
> $$z + 5 + G = R. \qquad \text{(C5)}$$

The constraint network for the equation constraints in the reformulated problem is given below. (The inequality constraints are not shown, but must, of course, be satisfied by a solution to the problem.)



A striking feature of this network is that it separates into three distinct components.

A solution to this problem using the ideas of domain-splitting and domain- and arc-consistency will be found using the following MATLAB function, which I've placed in the *S:\Cognitive Science\MATLAB\Cog Sci 242* directory.

```
function [ result ] = Lab4_CryptArithmetic()

% This function produces a (partial)solution to the crypt-arithmetic problem
% DONALD + GERALD = ROBERT
% as a constraint satisfaction problem by establishing domain and arc
% consistency for the constraint network for the problem.
%
% See the Lab 3, Lab 4, and Lab 5 handouts for information on interpreting
% 'result'.  If a solution is found, then 'result' contains the solution and
% the user is informed that it has been found. If only a partial solution
% has been found, the user is informed of that fact and 'result' contains
% the partial solution.
%
% This function also determines and reports its run time.
%
% This function calls the following functions with the syntax indicated:
%
%   Domain = Lab4_InitializeDomains()
%   ParameterArray = Lab4_InitializeParameters()
%   [failure, Domain] = Lab5_FirstComponent(Domain, parameters)
%   [failure, Domain] = Lab5_SecondComponent(Domain, parameters)
%   Domain = Lab4_ThirdComponent(Domain)
%   Lab4_OutputStatus(Domain)
%
% Author: Alan P. Knoerr          Last Update: September 28, 2015

% Initializations

tic                     % start the timing stopwatch

ParameterArray = Lab4_InitializeParameters();

failure_1 = true;

failure_2 = true;

k = 0;
```

```
% Prune domains to make component networks domain and arc consistent.

while (failure_2 == true)        % Parameters must work for both Components 1 & 2.

  while (failure_1 == true)      % Use Component 1 as an initial parameter screen.
    k = k + 1;
    parameters = ParameterArray(k,:);
    Domain = Lab4_InitializeDomains();  % Refresh 'Domain' for new parameters
    [failure_1, Domain] = Lab5_FirstComponent(Domain, parameters);
  end

  % Continue to Component 2 with the same parameter values working for
  % Component 1.

  [failure_2, Domain] = Lab5_SecondComponent(Domain, parameters);

  if (failure_2 == true)
     failure_1 = true;
  end

end

% Continue to Component 3 with same parameter values working for
% Components 1 & 2.

Domain = Lab4_ThirdComponent(Domain);

% Finish up and determine and report whether or not you have a solution
% to the whole problem, or just a consistent network.

Lab4_OutputStatus(Domain);

result = Domain;

% Report the run time for the function.

elapsedTime = num2str(toc);
disp(['This function took ', elapsedTime, ' seconds to run.'])

end
```

Some Housekeeping

Here are four of the functions called by **Lab4_CryptArithmetic**:

Domain = **Lab4_InitializeDomains**()

ParameterArray = **Lab4_InitializeParameters**()

Domain = **Lab4_ThirdComponent**(Domain)

Lab4_OutputStatus(Domain)

You should have written most of these last week in *Lab 4*.

Start MATLAB from the *All Programs* → *Cognitive Science* → *MATLAB Cog Sci 242* icon which, you'll recall, places you in your *G:\MATLAB* folder as your working directory, but also includes *S:\Cognitive Science\MATLAB\Cog Sci 242* in your path.

Verify that your versions of these functions are named as above and have the given syntax.

Be sure to test your functions. For those that depend on 'Domain,' create a suitable structure to use for testing. Refer to the *Lab 4, Part 1* handout for details on what these functions should do.

For example, to test **Lab4_ThirdComponent**, in the Command Window set

```
>> v = [ 1 2 3 4 6 7 8 9];
>> Domain = struct('A',  [1], 'B', [2], 'E', [3], 'G', [4], 'L', [6], 'N', [7], 'O', v, 'R',[ 8])
```

then see if **Lab4_ThirdComponent** behaves as expected.

CHECK POINT: Call me over if you have any questions about **Lab4_CryptArithmetic** or if you want me to take a look at any of your *Lab 4* functions.

The two functions we still need are

[failure, Domain] = **Lab5_FirstComponent**(Domain, parameters)

[failure, Domain] = **Lab5_SecondComponent**(Domain, parameters).

I've written **Lab5_SecondComponent** and placed it in
S:\Cognitive Science\MATLAB\Cog Sci 242.

Working in the MATLAB editor, **Open** the function **Lab5_SecondComponent** and **Save as ...**

Lab5_FirstComponent

in *your own G:\MATLAB* folder.

The reason for doing this is that you are going to write **Lab5_First Component** by modifying **Lab5_SecondComponent**.

First, however, you have to learn how **Lab5_SecondComponent** works.

Start by reading its *top-matter comments* (see the next page of this handout).

Note that it calls two of your constraint functions as well as two new functions,

Lab5_PruneDomain and **Lab5_NotEqualConstraints**.

Some More Housekeeping

Make sure that your constraint functions have the indicated syntax:

[failure, Constraint] = **Lab3_Constraint_C1**(L, R, w)

[failure, Constraint] = **Lab3_Constraint_C3**(B, N, R, x, y)

[failure, Constraint] = **Lab3_Constraint_C5**(G, R, z)

```
function [ failure, Domain ] = Lab5_SecondComponent( Domain, parameters )

% This function establishes arc consistency in the second component of the
% constraint network for the DONALD + GERALD = ROBERT crypt-arithmetic
% problem (as described in the Lab 4 handout), if this can be done
% with the nuisance parameter values in 'parameters'.
%
% If successful, it sets the 'failure' flag to 'false' and
% outputs 'Domain' with the arc consistent second component.
% Otherwise, it sets the 'failure' flag to 'true' and outputs the same
% 'Domain' it was given.
%
% See the Lab 5 handout for further details.
%
% This function calls the following functions with the indicated syntax:
%
% [failure, Constraint_C1] = Lab3_Constraint_C1(Comp2.L,Comp2.R,w)
% [failure, Constraint_C3] = Lab3_Constraint_C3(Comp2.B,Comp2.N,Comp2.R,x,y)
% [failure, Constraint_C5] = Lab3_Constraint_C5(Comp2.G,Comp2.R,z)
% [domain, pruned_flag] = Lab5_PruneDomain(domain,constraint)
% failure = Lab5_NotEqualConstraints(U)
%
% Author: Alan P. Knoerr          Last Upate: September 28, 2015
_____________________________________________________________________________

% Initialization

% Extract the second component from 'Domain' and values from 'parameters.'
% Note that all of the domains for this component start out being the same.

v = Domain.B;     % common initial domain

Comp2 = struct('B',v,'G',v,'L',v,'N',v,'R',v);

w = parameters(1);
x = parameters(2);
y = parameters(3);
z = parameters(4);

Var_Names = fieldnames(Comp2);   % Var_Names is a cell array of variable names
n_Names = length(Var_Names);
```

```
% Remove any values uniquely assigned to the first component
% from second component domains by first collecting any singletons from the first
% component into the vector U:

U = [];

if (length(Domain.A == 1))
  U = horzcat( U, Domain.A);
end

if (length(Domain.E == 1))
  U = horzcat( U, Domain.E);
end

% and then removing these values from the second component domains.

for i = 1:n_Names
     Comp2.(Var_Names{i}) =
               Comp2.(Var_Names{i})(not(ismember(Comp2.(Var_Names{i}),U)));
end

% Set "pruned" flags.

t = true;
pruned = struct('B', t, 'G', t,' L', t, 'N', t, 'R', t, 'R1', t, 'R3', t, 'R5', t);
```

Next read the *Initialization* section of this function (see the previous page & above).

Study it and the comments below. Consult MATLAB documentation online to answer questions you may have about different commands and functions used.

The main thing to note in the first part of this section is the use of *structures* both for *Comp1*, representing the domains for the first component variables, and for *pruned*, a set of logical flags that are going to be used to control the **while** loop that follows.

A *structure* stores information in *fields* (such as vectors) which are given *names*. The function **fieldnames** can be used to create a *cell array* (note the curly brackets used with *Var_Names* later on) that contains the names of the fields for the structure. In this case *Var_Names* is the cell array { 'B', 'G', 'L', 'N', 'R' }.

Cell arrays can be indexed, and the field names stored in them can be accessed in this way. Thus

Comp2.(*Var_Names*{3}) is the same thing as *Comp2.L*.

This makes it possible to use **for** loops when working with *structures*.

You'll also notice the line:

U = [];

This initializes the vector U to be an *empty vector* – a vector, but one that currently contains nothing.

The function **horzcat** is used to concatenate two row vectors.

The heart of the program is the **while** loop (see the next page).
Read this carefully now.

This cycles through the different constraints in Component 2 until either consistency is achieved or it is shown to be impossible to achieve.

The symbol || is the MATLAB operator for "logical or":

P || Q is true just in case at least one of P or Q is true.

The main thing to focus on here is the way in which the logical *pruned* flags are used to control passes through the **while** loop.

CHECK POINT: Call me over if you have any questions so far about what this program is doing and how it is doing it.

```
while ( pruned.B || pruned.G || pruned.L || pruned.N || pruned.R )

  % Prune the variables in the scope of Constraint C1 by first
  % updating constraint C1: set failure = true and return if not possible.

  [failure, Constraint_C1] = Lab3_Constraint_C1(Comp2.L,Comp2.R,w);

  if (failure == true)
    return
  end

  % Then prune the domains of L and R.

  [Comp2.L, pruned.L] = Lab5_PruneDomain(Comp2.L, Constraint_C1(:,1));
  [Comp2.R, pruned.R1] = Lab5_PruneDomain(Comp2.R, Constraint_C1(:,2));

  % Prune variables in the scope of Constraint C5 by first
  % updating constraint C5: set failure = true and return if not possible.

  [failure, Constraint_C5] = Lab3_Constraint_C5(Comp2.G,Comp2.R,z);

  if (failure == true)
    return
  end

  % Then prune the domains of G and R.

  [Comp2.G, pruned.G] = Lab5_PruneDomain(Comp2.G, Constraint_C5(:,1));
  [Comp2.R, pruned.R5] = Lab5_PruneDomain(Comp2.R, Constraint_C5(:,2));

  % Prune variables in the scope of Constraint C3, by first
  % updating constraint C3: set failure = true and return if not possible.

  [failure, Constraint_C3] = Lab3_Constraint_C3(Comp2.B,Comp2.N,Comp2.R,x,y);

  if (failure == true)
    return
  end

  % Then prune the domains of B, N and R.

  [Comp2.B, pruned.B] = Lab5_PruneDomain(Comp2.B, Constraint_C3(:,1));
  [Comp2.N, pruned.N] = Lab5_PruneDomain(Comp2.N, Constraint_C3(:,2));
  [Comp2.R, pruned.R3] = Lab5_PruneDomain(Comp2.R, Constraint_C3(:,3));

  % Determine whether the domain of R was pruned at all on this pass.

  pruned.R = pruned.R1 || pruned.R5 || pruned.R3;

end
```

Here's the last part of the **Lab5_SecondComponent** function. Given what you've already learned, it should be self-explanatory.

```
% Check "not equal" constraints for singleton domains in the second
% component by first collecting the singletons into the vector U,

U = [];

for i = 1:n_Names
   if length( Comp2.(Var_Names{i}) ) == 1
     U = horzcat( U, Comp2.(Var_Names{i}) );
   end
end

% Then check the "not equal" constraints.

failure = Lab5_NotEqualConstraints(U);

if (failure == true)
  return
end

% If you get this far, arc consistency has been achieved for this component.
% Update Domain with this consistent second component.

for i = 1:n_Names
   Domain.(Var_Names{i}) = Comp2.(Var_Names{i});
end

end              % This is the end of  the function.
```

>> **type Lab5_PruneDomain**

and

>> **type Lab5_NotEqualConstraints**

to find out what these functions are doing and how they're doing it.
They're both short.

CHECK POINT: Any further questions about these last two functions or anything else about **Lab5_SecondComponent**?

Now, YOU are going to write **Lab5_FirstComponent** by editing **Lab5_SecondComponent** (which, hopefully, you now understand).

Save your work on your G: drive.

To guide you, I've placed a "comment" outline of the program on the next pages.

When you are finished, you will be able to run **Lab4_CryptArithmetic**!

function [failure, Domain] = L**ab5_FirstComponent**(Domain, parameters)

% This function establishes arc consistency in the first component of the
% constraint network for the DONALD + GERALD = ROBERT crypt-arithmetic
% problem (as described in the Lab 4 handout), if this can be done
% with the nuisance parameter values in 'parameters'.
%
% If successful, it sets the 'failure' flag to 'false' and
% outputs 'Domain' with the arc consistent first component.
% Otherwise, it sets the 'failure' flag to 'true' and outputs the same
% 'Domain' it was given.
%
% See the Lab 5 handout for further details.
%
% This function calls the following functions with the indicated syntax:
%
% [failure, Constraint_C2] = **Lab3_Constraint_C2**(Comp1.A,Comp1.E,w,x)
% [failure, Constraint_C4] = **Lab3_Constraint_C4**(Comp1.E,y,z)
% [domain, pruned_flag] = La**b5_PruneDomain**(domain,constraint)
% failure = **Lab5_NotEqualConstraints**(U)
%
% Author: Last Update:

% Initializations

% Extract the first component from 'Domain' and values from 'parameters.'
% Note that all of the domains for this component start out being the same.

% common initial domain

% Set "pruned" flags.

while (pruned.A || pruned.E)

% Prune the variable E in the scope of Constraint C4 by first
% updating constraint C4: set failure = true and return if not possible.

```
    % Then prune the domain of E.


    % Prune the variables in the scope of Constraint C2 by first
    % updating constraint C2: set failure = true and return if not possible.


    % Then prune the domains of A and E (in principle).


    % Determine whether the domain of E was pruned at all on this pass.

end

% Check "not equal" constraints for singleton domains in the first component
% by first collecting the singletons into the vector U.

                                % Var_Names is a cell array of variable names


% Then check the "not equal" constraints.


% If you get this far, arc consistency has been achieved for this component.
% Update Domain with this consistent first component.


end                             % end of the function
```

When your **Lab5_FirstComponent** function is finished and has run successfully as part of **Lab4_CryptArithemtic**, upload it to its Moodle submission site. Be sure to include your name(s) and the date as part of the top-matter comments, and to include the comments I outlined above.

Performance and Reflection

You should test your function after you've written it. I've tested all the functions that I've written and have been able to get my own **Lab5_FirstComponent** to work on its own as well running in **Lab4_CryptArithmetic**.

When everything is working, run the programs indicated below and complete the table:

Program	*Run Time*
Lab2_CryptArithmetic	
Lab3_CryptArithmetic	
Lab4_CryptArithmetic	

You'll see that there's a significant improvement as we've progressed through the labs. I think you'll find that the run times have improved by an *order of magnitude* in each case! (Make sure you know what this means.)

Run time is one thing programmers try to minimize.

The other is memory usage. One indicator of this is the size of the arrays that are created while running the program.

For each of these programs, identify the size (dimensions) of the major arrays created while running the program. Be sure to also consider arrays created by functions the main program may call. (Note, however, that after those functions finish, any arrays they created while running no longer exist.)

Program	*Largest Array Sizes*
Lab2_CryptArithmetic	
Lab3_CryptArithmetic	
Lab4_CryptArithmetic	

Collecting Visual Search Data

In the second part of today's lab you will gain some hands-on experience with a visual search experiment we will soon be studying in depth.

Each member of a lab group needs to do this themselves!

To begin, create a new folder G:\CogLab on your G: drive.

Then run the program **CogLab.exe** found in this directory:

S:\Cognitive Science\CogLab.

A menu of activities will appear on the screen.

Choose *Visual Search.*

Carefully read the instructions and then run the experiment. BE SURE TO INCLUDE YOUR NAME – and KEEP GOING! You are collecting data on your own visual search behavior. There are close to 100 trials, but they go quickly.

When you are finished, save the results in both HTML and CogLab formats in your *CogLab* folder on your G: drive.