We have been looking at the following 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 when the letters have been replaced by their assigned numbers, the following equation is satisfied: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;
>
> and 5 is assigned to the letter D.

In *Lab 3* we reformulated the problem 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 the following *inequality constraints*:
>
> $$\begin{array}{cccc} A \neq B, & A \neq E, & A \neq G, & A \neq L, \quad A \neq N, \quad A \neq O, \quad A \neq R, \\ & B \neq E, & & \cdots \qquad\quad B \neq R, \\ & & E \neq G & \cdots \qquad\quad E \neq R, \\ & & \ddots & \cdots \qquad\quad \vdots \\ & & & N \neq O, \quad N \neq R, \\ & & & O \neq R; \end{array}$$
>
> and the following *equation constraints*:
>
> $$1 + 2L = w * 10 + R, \qquad \text{(C1)}$$
>
> $$w + 2A = x * 10 + E, \qquad \text{(C2)}$$
>
> $$x + N + R = y * 10 + B, \qquad \text{(C3)}$$
>
> $$y + O + E = z * 10 + O, \text{ and} \qquad \text{(C4)}$$
>
> $$z + 5 + G = R. \qquad \text{(C5)}$$ ( We already know that D = 5 and T = 0. )

In *Lab 3* we began to consider an algorithmic solution inspired by the way a human problem-solver would use these constraints to dramatically limit the assignments that would have to be checked.

This approach was based on regarding the reformulated problem as a **constraint satisfaction problem** (CSP) consisting of

- a set of variables,
- a domain for each variable (its set of possible values), and
- a set of constraints.

A solution for the problem is an assignment of values to variables that satisfies the constraints.

By the end of *Lab 3,* the following programs had been written. They each output a *Constraint* array representing their respective inequality or equality constraint.

[ failure, Constraint ] = **NotEqualConstraint**( X, Y )

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

[ failure, Constraint ] = **Lab3_Constraint_C2**( A, E, w, x )

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

[ failure, Constraint ] = **Lab3_Constraint_C4**( E, O, y, z )

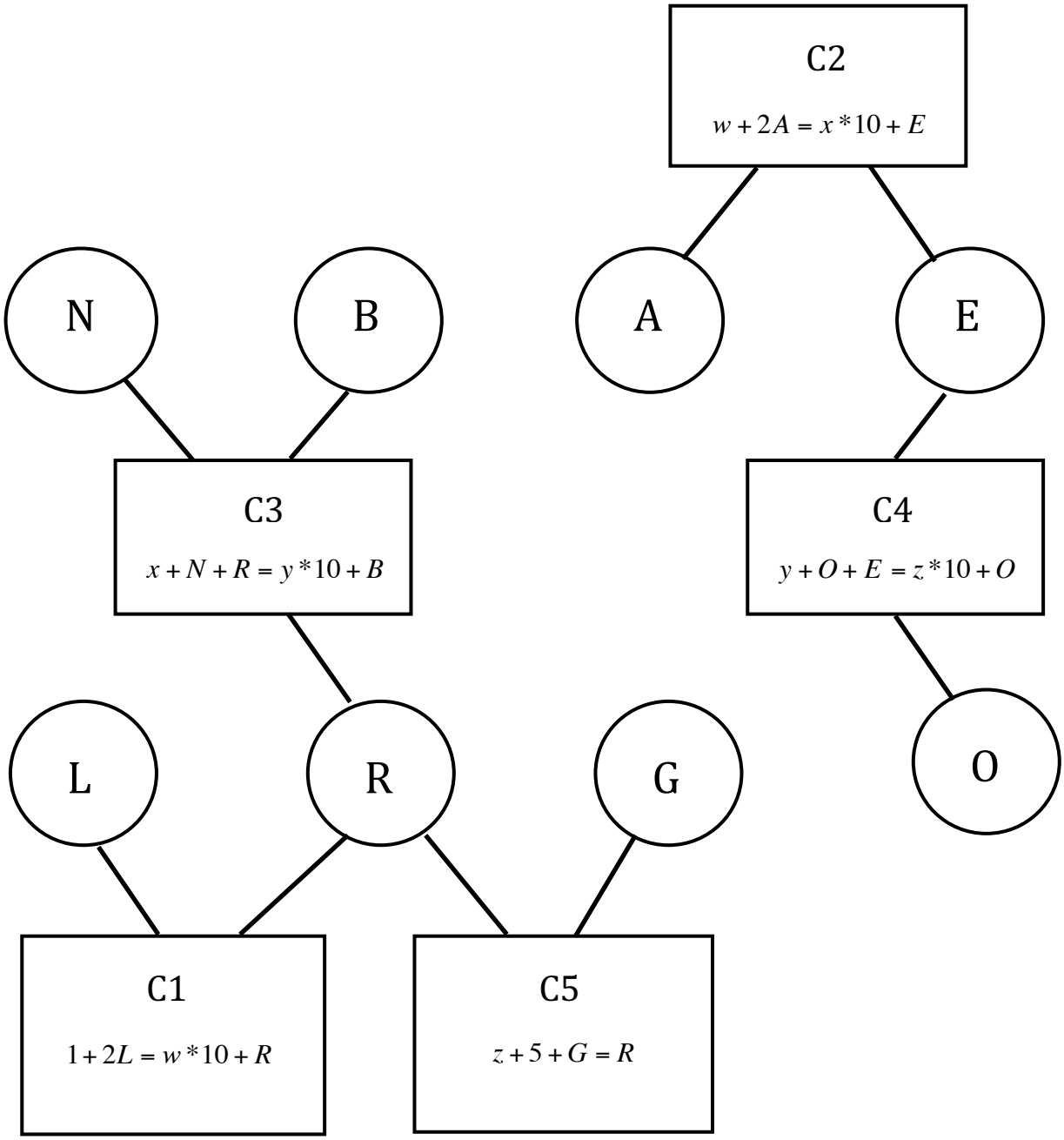
[ failure, Constraint ] = **Lab3_Constraint_C5**( G, R, z )

In these functions, the capitalized *input variables* are *row vectors* representing the domains of the problem variable having the same name. The *nuisance parameters* are *scalars* (numerical values) assigned at the time the function is called. The output *Constraint* is an *array*, each *row* of which is a tuple of values drawn from the given domains for problem variables in the scope of the constraint that satisfies the constraint when any nuisance parameters have their assigned values. The other output variable, *failure*, is a logical flag that is **true** if the function failed to find any tuples satisfying the constraint and **false** otherwise.

For example, if the tuple ( 6, 3 ) appears in the *Constraint* array output by **Lab3_Constraint_C1** when *w* is assigned the value 1, this means that 6 is in the current domain of L, 3 is in the current domain of R, and that L = 6, R = 3 satisfies the constraint $1 + 2L = w * 10 + R$ when w = 1.

A basic operation in the algorithm we will develop to fully solve the crypt-arithmetic problem is **pruning the domain of a variable to achieve "arc consistency**."

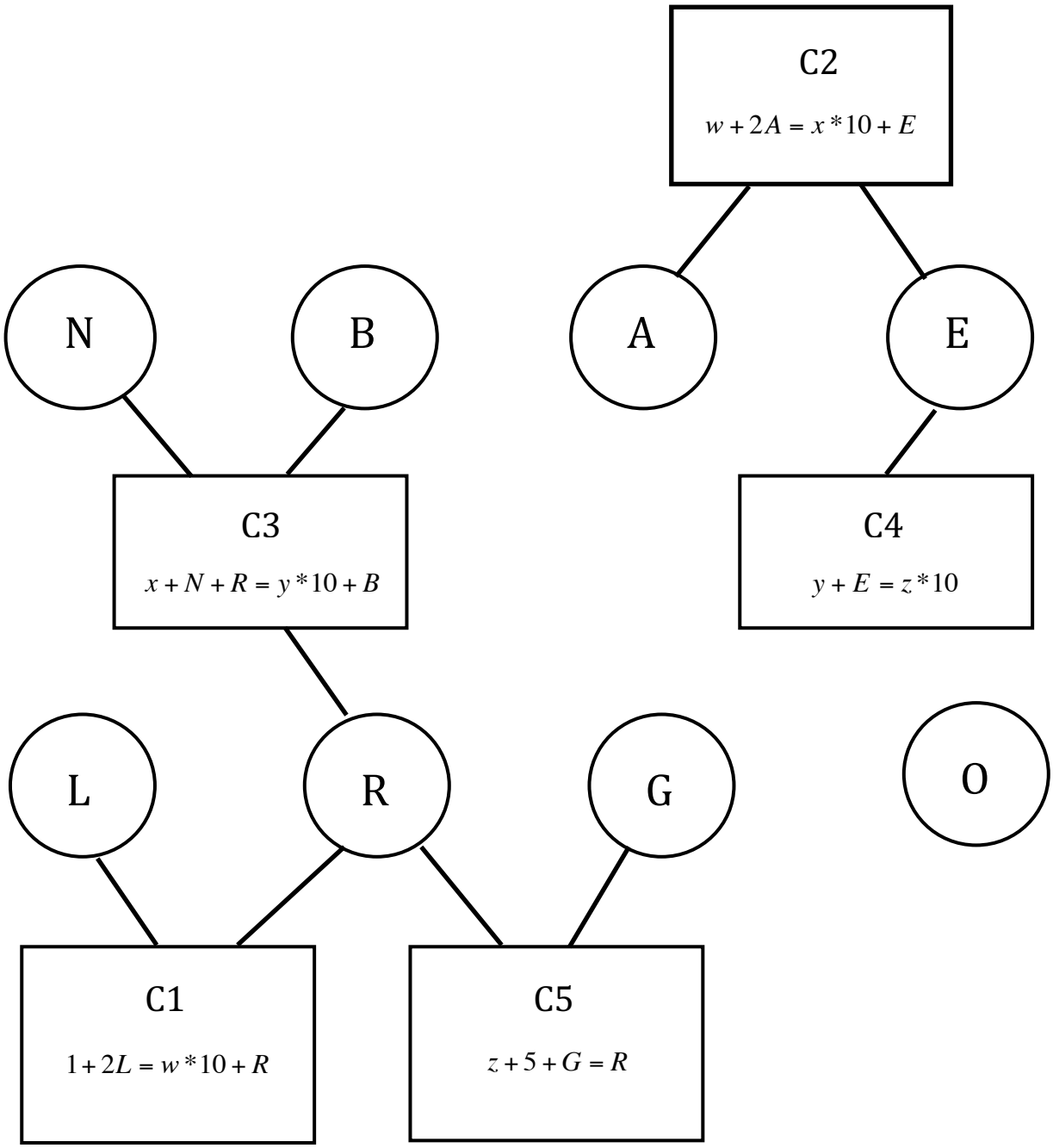
Recall that a CSP can be represented graphically by a constraint network. The portion of **the constraint network for the reformulated problem** involving the equation constraints is shown below (with the domains suppressed):



Note, however, that constraint C4 actually has no effect on the variable O since

$y + O + E = z*10 + O$ is algebraically equivalent to $y + E = z*10$.

If we use this observation to simplify constraint C4, then the portion of the constraint network representing the equation constraints becomes



**Now start MATLAB.**

**Edit your Lab3_Constraint_C4 function** to reflect the change to constraint C4 just mentioned above. The syntax of your function should now read:

Constraint = **Lab3_Constraint_C4**( E, y, z )

and the function should output *Constraint* as a 1-by-1 array containing the unique value in the domain of E satisfying

$$y + E = z * 10, \text{ given } y \text{ and } z,$$

if this value exists.

Edit your comments to reflect this change (and the date you made the change).

While you're editing this program, take a minute to make sure you understand how the *failure* flag is working:

Note that the output index *k* is initialized to 0, and is only increased when a tuple is found that satisfies the constraint. If no such tuple is found by the time the function exits all the **for** loops, then *k* will *still* have the value 0. Otherwise it will have a positive value greater than or equal to 1. With these points in mind, **explain what the following MATLAB code does**:

```
failure = ( k == 0 );

 if  ( failure == true )     % if no tuples satisfy constraint
     Constraint = [ ];       % then set Constraint to be empty
     return
else
    Constraint = Constraint(1:k, : );
end
```

**CHECK POINT**: Call me over to explain to me why we want a failure flag in these functions – or to ask me why, if you're uncertain about this!

Domain and Arc Consistency

We are almost ready to put together an algorithm to solve the crypt-arithmetic problem. However, we first need to be explicit about how to use constraints to eliminate possible values for a variable. (This is called "pruning its domain.")

We will base our domain pruning on two concepts:

Let $\langle V, c(V) \rangle$ denote an arc
connecting variable node $V$
to a constraint node $c$
depending *only* on $V$
( a **unary constraint**).

Such an arc is **domain consistent**
if for each value $v$ in the domain of $V$,
the constraint $c$ is satisfied when $V = v$.

Refer to our latest constraint network for the problem, the one with the modified constraint C4. There is one unary constraint in this network. (Assume the nuisance parameters have been given values.)

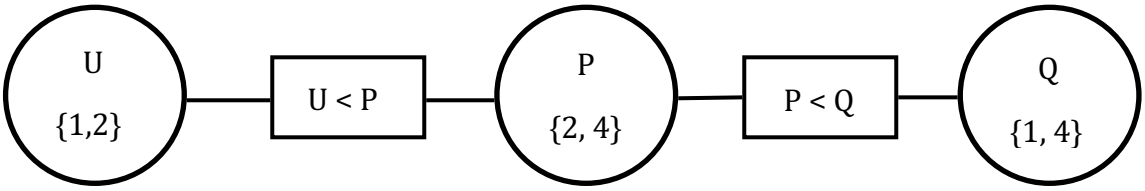
Which constraint is it? What would it mean to say that the arc from variable E to that constraint was domain consistent?

Let $\langle U, c(U, V_1, V_2, \ldots, V_m) \rangle$
denote an arc connecting variable node $U$
to constraint node $c$
having scope $U, V_1, V_2, \ldots, V_m$.
(a **multivariate constraint**).

Such an arc is **arc consistent**
if for each value $u$ in the domain of $U$,
there are *some* values $v_1, v_2, \ldots, v_m$ in the domains of $V_1, V_2, \ldots, V_m$, respectively,
such that constraint $c$ is satisfied
when $U = u$ and $V_1 = v_1, V_2 = v_2, \ldots, V_m = v_m$ .

This is a lot to take in! The notation and careful language needed to specify this concept precisely is best read in small bites (that's why I've written it using many lines). However, it's probably easiest to gain an understanding of the concept by working through an example.

*Example*: Consider the following constraint network:




The arc from variable node U to constraint node U < P is arc consistent. Why?

The arc from variable node P to constraint node P < Q is NOT arc consistent.
Why not?

**CHECK POINT**: Call me over to check your answers to the last three questions about domain and arc consistency.

A *constraint network* is **arc consistent** if *all* of its arcs to *any* unary constraint is domain consistent, and *all* of its arcs to *any* multivariate constraint is arc consistent.

If we have the right "carry digits" (the right nuisance parameter values) and we prune the domains of the main variables in the problem to make the constraint graph based on the five equation constraints arc consistent, then we will either have solved the problem or greatly reduced the number of solution candidates that need to be checked explicitly.

Pruning variable domains to achieve arc consistency sounds complicated, but it will actually be very simple in MATLAB because of the way we've chosen to represent the constraints.

Remember that each constraint is an array where each column contains values drawn from the domain of only one variable. A given value *v* for variable a V is part of a tuple satisfying the constraint just in case it appears in the *Constraint* array column for that variable.

Suppose *Constraint* is a constraint array for a constraint with variable V in its scope, and suppose the column for variable V is column *k*.

Also suppose that V, in MATLAB, denotes a row vector serving as the current domain of variable V.

Then in MATLAB:

Constraint(:, k)

selects column *k* from the array *Constraint*;

**ismember**( V, Constraint(:, k) )

creates a logical vector the same length as V
with a logical 1 in the same position as each value in V that is in *Constraint*(:, k),
and a logical 0 elsewhere;

V( **ismember**( V, Constraint(:, k) ) )

selects *only* those values from the domain vector V
with a logical 1 in the same position as each value in V that is in *Constraint*(:, k);

and finally,

V = V( **ismember**( V, Constraint(:, k) ) )

replaces the *original* domain vector V with one that has been *pruned* to make the arc from V to the constraint consistent. Note that no **for** loops or **if** statements were needed to accomplish this!

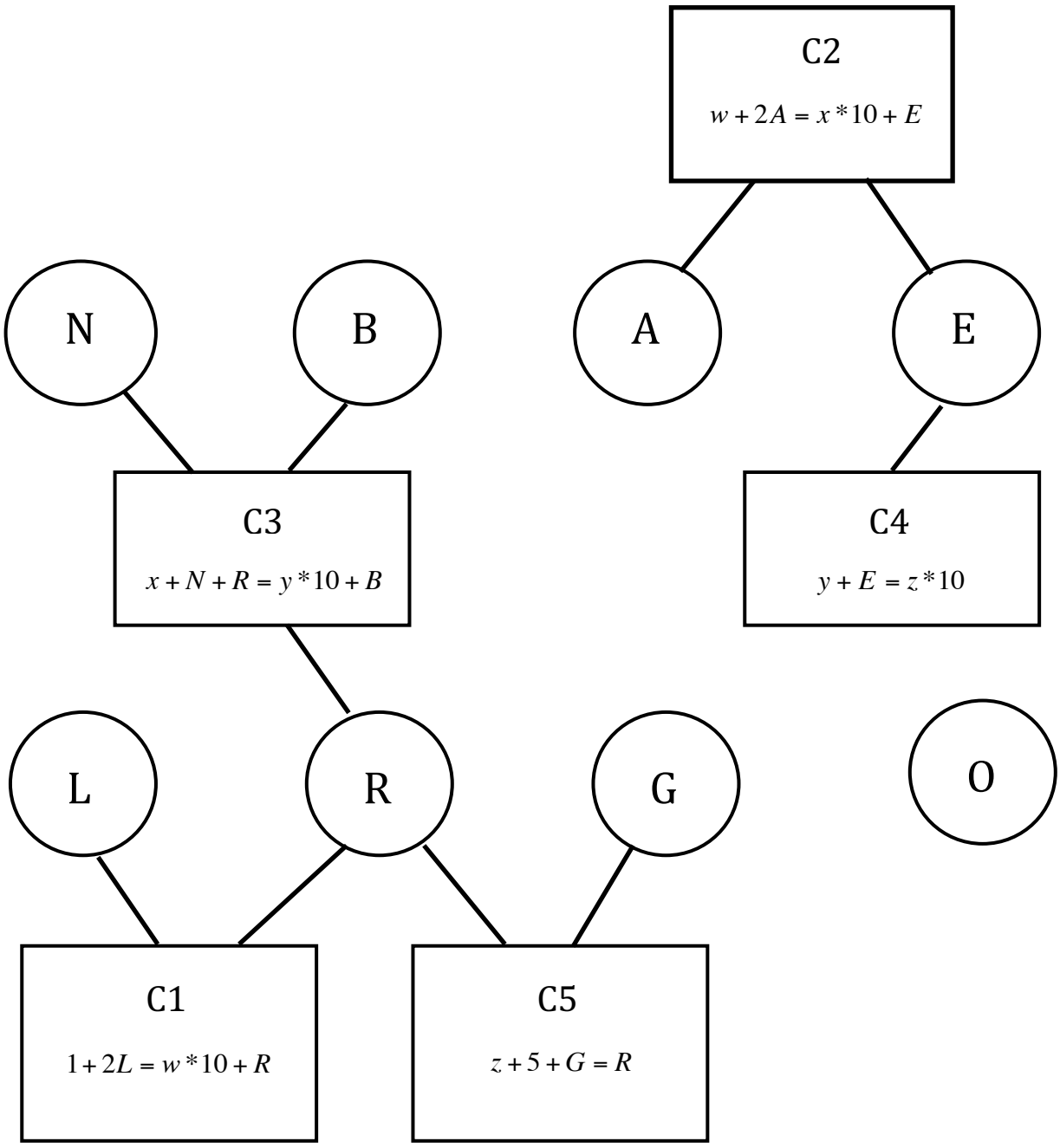
**Try this now yourself.** Execute the following MATLAB commands in the *command window*, predicting what each one will do before executing it. So you can see the results at each step, **DON'T put semicolons at the end of each line**.

```
>> U = [ 1, 2, 3 ]

>> V = [ 1, 2, 3 ]

>> Constraint = [ 1  2; 1  3; 2  3]     % this is the array for the constraint U < V

>> ismember( U, Constraint(:, 1) )

>> U = U( ismember( U, Constraint(:, 1) ) )

>> V = V( ismember( V, Constraint(:, 2) ) )
```

*Selecting* elements from a vector or array using a *logical vector or array* is a technique called *logical indexing*. It can be very powerful, and we will make free use of this programming technique from now on.

Writing a Consistency Algorithm

Let us once more examine the constraint network comprising the equation constraints:



We can see that this constraint graph has three connected components.

*First Component*

- The first component involves variables A and E, constraints C2 and C4, and all the nuisance parameters: *w*, *x*, *y* and *z*.
- Moreover, C4 is a *unary* constraint. Therefore,
    - given values for *y* and *z*, the value for E is *uniquely determined* by domain consistency for the ⟨E, C4⟩ arc;
    - and given values for *w* and *x*, along with the unique value for E, the value for A will be *uniquely determined* by arc consistency for the ⟨A, C2⟩ arc.

  We know these values will be *unique* because the constraints are *equality* constraints.

  Moreover, since *all* the nuisance parameters are involved in this section, we should be able to rule out many possible assignments of values to them by concentrating first on this component.

*Second Component*

- The second component involves the variables B, G, L, N and R as well as the constraints C1, C3 and C5 and, again, all the nuisance parameters.
- Constraints C1 and C5 are *binary*, constraint C5 is *ternary* (i.e. involves three variables), R is the only variable in the scope of more than one constraint, and R is in the scope of *all* the constraints. Therefore, to prune the variable domains quickly we should *cycle* through the arcs in the following order, establishing arc consistency in each case as we come to it:
    - first make the binary arcs ⟨L, C1⟩ and ⟨G, C5⟩ consistent;
    - then make the arcs involving R consistent: ⟨R, C1⟩, ⟨R, C3⟩, ⟨R, C5⟩;
    - then make the other arcs in the ternary constraint consistent: ⟨N, C3⟩ and ⟨B, C3⟩.
- If we run through a complete cycle in this order without pruning any of the variable domains further, we'll know this *entire component* is arc consistent.

*Third Component*

- This consists of the variable O and no constraints. By default it is already arc consistent.

**CHECK POINT**: In the preceding discussion about the order in which to examine arcs in the constraint network, two important claims were made:

*Claim 1*: Given values for the nuisance parameters, the variables A and E in the first component will have unique values once domain and arc consistency are established.

*Claim 2*: If we complete a cycle through the arcs in the second component in the given order with no further changes to the domains, then we know that the second component is arc consistent.

**Call me over and explain to me why both these claims are true.**

We are now ready to write an algorithm to make the constraint graph consistent.

### *The Consistency Algorithm*

Initialize variable domains and possible nuisance parameter configurations.

For each possible nuisance parameter configuration:

- *First Component*
  - Establish domain and arc consistency for the first component.
  - If this can't be done, discard that nuisance parameter configuration and choose another one. (Start again on the first component with this new configuration.) Repeat until consistency has been established.

- *Second Component*
  - If consistency of the first component has been established for a particular nuisance parameter configuration, prune the values already assigned to A and E from the domains of the remaining variables.
  - Then establish arc consistency for the second component.
  - If this can't be done, discard the current nuisance parameter configuration and choose another one. (Start over again on the first component with this new configuration.)

- *Third Component*
  - If consistency of the second component has been established for a particular nuisance parameter configuration, *and* has resulted in a unique assignment of values to each variable in that component, remove those values from the domain of O. In this case, we have found a solution to the crypt-arithmetic problem. Otherwise, we have only achieved arc consistency for the constraint network and further work is needed to solve the crypt-arithmetic problem.

Back in the Command Window of MATLAB,

**>>type Lab4_CryptArithmetic**

Compare the listing of this program with the algorithm outlined above.

Be sure you can identify *which parts of the program* are carrying out *which parts of the algorithm*. Some details of the algorithm are carried out by functions you will soon be writing.

**CHECK POINT**: Call me over to explain what **while** is doing in this program.

**Lab4_CryptArithmetic** depends on a number of other functions:

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 )

In the next section of this lab you are going to write some of these functions – the **Lab4** ones. Save them to your G: \MATLAB folder as you write them.

In each case, pay attention to the computational theory for the function and my suggestions for algorithm, mechanism and implementation.

Domain = **Lab4_InitializeDomains** ( )

*Computational Theory*

This function initializes the domains for the main variables.

All these domains should be set to the same set of values initially. Execute the following in the Command window:

>> v = [1, 2, 3, 4, 6, 7, 8, 9]

However, because we will be pruning the different domains differently we need a way of naming and storing the domains in such a way that we can easily retrieve and operate on them individually.

*Mechanism Notes*

The MATLAB data structure we'll use for this purpose is called, simply, a "structure."

*Implementation Notes*

The following command will create the structure *Domain*.

Do this now in the Command window in MATLAB:

>>Domain = **struct**('A', v, 'B', v, 'E', v, 'G', v, 'L', v, 'N', v, 'O', v, 'R', v)

The variable names in quotes are called the "fields" of the structure, and each field is immediately followed by its "value." (In this example, all fields have the same value.)

To refer to or to retrieve the value of a field, simply call the structure name, followed by a period, followed by the field name.

Do this now in MATLAB:

>>Domain.A

**Now that you understand how to create a structure, follow the guidance below to write the Lab4_InitializeDomains function. Be sure to comment it appropriately.**

ParameterArray = **Lab4_InitializeParameters**()

*Computational Theory*

Each of the nuisance parameters has two possible values: 0 or 1.
There are four of these, so there are $2^4 = 16$ different configurations of nuisance parameter values.

*Algorithm*

Create a 16-by-4 array of zeros.
Then put each configuration into this array directly. While it is possible to use a loop to do this, this is not necessary provided you organize your list of possible configurations in such a way that you are certain no possibilities have been missed.

*Mechanism & Implementation Notes*

Consult MATLAB help for how to work with the **zeros** function.

Assuming *ParameterArray* has been initialized as an array of zeros, you would put the first two configurations into this array as follows:

```
ParameterArray( 1, : ) = [0 0 0 0];
ParameterArray( 2, : ) = [1 0 0 0];
```

etc.

Domain = **Lab4_ThirdComponent**( Domain )

*Computational Theory*

This function identifies the domain values which have already been assigned to other variables and removes them from the domain of the variable O.

*Algorithm Notes*

A value hasn't definitely been assigned to a particular variable unless that value is the *only* value in that variable's domain. So simply check the domain of each of the other variables domain to see if it contains just one value, and if so, remove that value from the domain of O.

*Mechanism & Implementation Notes*

Remember that each of the domains in the structure *Domain* is a row vector.

You can use the **length** function to find the lengths of a domain vector, which will tell you how many values are in the corresponding domain.

Use the **ismember** function and logical indexing to remove values.
For example, we can remove from Domain.O the values that are in Domain.A using

Domain.O = Domain.O( **not**( **ismember**(Domain.O, Domain.A) ) );

(Experiment with this construction in the *Command Window*, working from the inside out, until you understand *exactly* what it is doing. Then use this technique in your program.)

Note: The other "component" functions are more involved;
we'll tackle those next week.

**Lab4_OutputStatus**( Domain )

*Computational Theory*

At this point we can see how the strategy of domain pruning to achieve network consistency can reduce the number of values in each domain, but we don't know whether it will leave only one value in each domain, i.e. produce a solution.

This function checks the pruned *Domain* and makes that determination.

In the event that a solution has been found, it should *display* a message saying so.

In the event that a solution has not been found, it should *display* a message saying that.

*Algorithm Notes*

A solution has been found when each variable's domain contains exactly one value. So simply check the domain of each of variable to see if it does contain just one value. *As soon as* a domain has been found containing more than one value, you know that a solution has *not* been found.

*Mechanism & Implementation Notes*

Remember that each of the domains in the structure *Domain* is a row vector.

Use the **length** function to find the lengths of those vectors.

WRAPPING UP

At the conclusion of this lab, you should upload a copy of your **Lab4_ThirdComponent** function to the appropriate Moodle folder. Be sure that the uploaded version includes the initials of all your lab partners in the file name. Also be sure that your function is commented adequately.

*Note*: If you were not able to finish all the programming for this lab today, arrange with your lab partners to have this work completed by next week's lab.