A search methodology for finding a feasible solution to a system of equations.

Eg: Given an $n \times n$ grid, with some entries given, fill the remaining squares so that each row and column contain all numbers from 1 to $n$ so that each number appears exactly once in each row and each column.

Eg: 

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Solution:

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>4</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Forced:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Immediately:

- Only way to place these entries in these rows.

Now the "4" is forced.
How did we find this?

Can determine which numbers can go in each square:

```
  1  2  3
  2  3  4
  4  1  2
```

Only possibility for a "2" in this row.

So this is fastest.

Once the "4" is set, update the other empty squares:

Algorithm:
0. If no empty squares, STOP.
1. For each empty square, find the possible entries.
2. If any square has a unique choice, make that choice.
   Return to 1.
3. Else, if any number can only appear in one particular position in a row or column, make that choice.
   Return to 1.
4. Else, try having the entry with the smallest number of choices.
   Repeat for the steps 0, 1, 2, 3 to try to find a solution.
5. If end up infeasible, back track, add the constraint that can't make the last choice.
Example when nothing is forced, and making a wrong decision leads to infeasibility:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

If put '1' here, then can't put '1' in the last row or column.

Example used for optimization:

This "finding numbers" example uses the 'all different' constraint, which is easy to write in constraint programming but harder to write in integer programming.

Constraint programming can be used for optimization:

1. Find a feasible solution.
2. Impose a constraint that must do better than the current feasible solution, and return to 1.

If Step 1 fails, return the last feasible solution found.
Eg: Scheduling problem:

Job shop problem: See lecture.

Have several machines: 1, ..., nMachines.

Have several jobs: 1, ..., nJobs.

For each job, need to complete several tasks: 1, ..., nTasks. Tasks must be completed in order.

A given task on a particular job can be completed on some subset of the machines, and takes a certain duration, \((resource[j,k])\).

Each machine can perform at most one task at a time.

The makespan is the time the last job finishes.

The objective is to minimize the makespan.

Constraints:

1. For all \((j \in \text{Jobs})\), task \([j, n \text{Tasks} - 1] \) precedes, makespan.

   Ensures Melanie is calculated accurately.

2. For all \((j \in \text{Jobs})\),
   
   For all \((e = 1, ..., n \text{Tasks} - 1)\),
   
   Task \([j, e]\) precedes task \([j, e + 1]\).

3. For all \((j \in \text{Jobs})\),
   
   For all \((e \in \text{Tasks})\),
   
   Task \([j, e]\) requires task \([\text{resource}[j, k]]\).

4. Unique Resource \(\text{tool}[\text{Machines}]; \quad \text{each machine can only be used for one task at a time.}\)