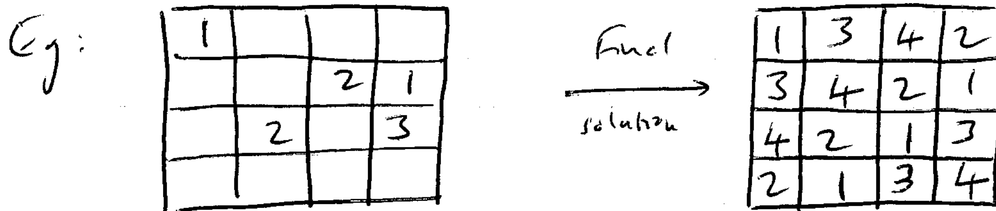
Constraint Programming

(Lustig & Puget, Interfaces 31:6, 2001, pp ~~77~~ 29-53.

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

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

Eg:



Forced

1			2
		2	1
4	2	1	3
2	1		

immediately: ~~only choice for these squares.~~ only way to place these entries in these rows

Now the "4"s get forced

How did we find this?

Can determine which numbers can go in which squares:

1	34	34	24
34	34	2	1
4	2	14	3
234	134	134	24

Only possibility for a "2" in this row.

So this is forced

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 fixing the entry with the smallest number of choices.
~~Explore for~~
Use steps ①,②,③ to try to find a solution

⑤ If end up infeasible, backtrack,
add the constraint that can't make the last choice.

Eg An example where nothing is forced, and making a wrong decision leads to infeasibility:

1	2		
	1	2	
2		←	
			2

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

~~Can be used for optim~~

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 ~~constraint~~ solution
2. Impose a constraint that must do better than this feasible solution, and return to (1).

If Step (1) fails, return the last feasible solution found.

Eg: Scheduling problem:

Job shop problem: See handout.

Have several machines: 1, ..., nbMachines

Have several jobs: 1, ..., nbJobs

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

~~Eg~~ 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.

Get constraints:

① forall (j in Jobs)
    task[j, nbTasks] precedes MAKESPAN;

Ensures makespan is calculated accurately.

② forall (j in Jobs)
    forall (t in 1..nbTasks - 1)
        task[j, t] precedes task[j, t+1]

③ for all (j in Jobs)
    for all (t in Tasks)
        task[j,t] requires tool[resource[j,t]];

④ UnaryResource tool[Machines];

each machine can only be used for one task at a time.

This is the hard constraint to express with integer programming