Modeling:

- _ each cell is a variable
- Domain of each variable is { empty, positive, negative}
- Constraints:

In Code, "not-initialized" is in domain of every voriable

and at first (before any assignment) all

variables are "not-initialized".

Pole consistency of for every
$$X_{ij}$$
 —, if $|X_{ij}| = positive$, then $|X_{ij+1}, X_{i+1}|$ can't be positive

$$|X_{ij}| = positive$$

$$|X_{ij}| = positive$$

$$|X_{ij+1}, X_{i+1}|$$

$$|X_{ij}| = positive$$

$$|X_{ij}| = posit$$

Row consistency:

Column Gusistency:

Implementation of backtrack_solve() function is based on this pseudocode:

Backtracking search

function BACKTRACKING-SEARCH(csp) return a solution or failure **return** RECURSIVE-BACKTRACKING({}, csp)

```
function RECURSIVE-BACKTRACKING(assignment, csp) return a solution or
  failure
  if assignment is complete then return assignment
  var \leftarrow SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
       if value is consistent with assignment according to CONSTRAINTS[csp] then
                add {var=value} to assignment
                result \leftarrow RRECURSIVE-BACTRACKING(assignment, csp)
                if result ≠ failure then return result
       remove {var=value} from assignment
  return failure
```

```
public static void backtrack_solve(Csp csp) {
    boolean temp = recursive(csp);
    System.out.println(temp);
public static boolean recursive(Csp csp) {
    recursion count simple backtracking++;
if(csp.isComplete()) {
        System.out.println("assignment is complete");
                                                                   selecting
        csp.printBoard( x: -1, y: -1);
        return true;
                                                        variable do assign is
                                                                              bused
    Variable varToAssign = null;
        boolean stop = false;
for(int j = 0; j < csp.m; j++) {
  if(csp.vars[i][j].value == VarState.notInit) {
                 varToAssign = csp.vars[i][j];
                 break;
    for (Varstate val: vartoAssign.domain) { > Sclessing ratue to assign
                                                                                 apploton
        csp.vars[varToAssign.row][varToAssign.col].value = val;
        if(csp.vars[varToAssign.row][varToAssign.col].isConsistent(csp)) {
            boolean result = recursive(csp);
        csp.vars[varToAssign.row][varToAssign.col].value = VarState.notInit;
```

Implementing Forward Checking:

- Source book mentions that Forward Checking only checks Binary Coustaints, but in this problem, checking higher consistency is not very costly, so I wrote & forward checking functions:

-Binary Forward Cheeking: only checks birrary constraints
(pair consistency f
pole consistency)

all forward checking functions veturn
List of previous domain

6f variables that Their domain's
Changes, because we may need to
restore previous domain when
we back drack.

- _Forward Checking(): same indemontation, except it also checks row/column consistency
- _ abnormal Forward Checking () & same implementation, it can also removes "empty" from domain

Implementing MRV, LCV:

LCV: using Forward Checking, update domains with respect to assigning a varStrate, then counts the "flexibility" and gees to next var strate, at the end returns 5 orted list of legal moves of the variable we choose to assign

The end returns 1 set of legal moves of the variable we choose to assign

(first Varstute is the least

MRT: chose variable with most remaining value, Forward Checking updates the Lomain (legal moves).

* effectiveness of MRV: *

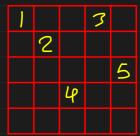
In this problem, using MRT has regative effect compare to chosing variables bused on their index. When choosing variable to assign bused on their index (from left top to right bottom) we iterate over cells of one row and then yo to the next

Vow, hearing that when back rack is in the second vew, it guarantees that Row postney constraints on first row are satisfied. (it makes sure that it satisfies current row constraints before moving to next vow).

But using MKV, we might shows e a variable in 1-th row, and next variable that it chooses to assight is in (1+k)-th column.

\vdash	2	3	4	5
6	7	8	う	1
11	12	1]	14	15
16	17	18	19	20
21	22	23	2. Y	25

Order of assighment in Index-based



assignment in MET

Implementing Arc-consistency:

Implementation of ac3() function is based on this pseudocode:

function AC-3(csp) returns false if an inconsistency is found and true otherwise $queue \leftarrow$ a queue of arcs, initially all the arcs in cspwhile queue is not empty do $(X_i, X_j) \leftarrow POP(queue)$ if $REVISE(csp, X_i, X_j)$ then

if size of $D_i = 0$ then return falsefor each X_i in X_i . NEIGHBORS - $\{X_j\}$ do

add (X_k, X_i) to queuereturn truefunction $REVISE(csp, X_i, X_j)$ returns true iff we revise the domain of X_i $revised \leftarrow false$ for each x in D_i do

if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then delete x from D_i $revised \leftarrow true$ return revised

Comparing Arc-consistency with other heuristic humous:

- Arc-consistency only deals with Binary Constraints and does not check Row/G/ Constraints because beguve not bornary, So its not Kint swong.

- ac3() function has a large overhead and it's very (OSTLY.

_although Arc Consistency is (Stronger) than Binary Forward Checking, it does not

with binary Forward Checking ()

run fastor, with acs()

for example for test2.txt number. 2178

recoursins

run: 798 ms

time

7969

(for example every since 10 new vourinbles gets assigned)

59 ms

- it's obvious dust Arc-Consisteng CAN be weaker than abnormal Forward Checking because it does not check Raw/C/ consistency -it's probably bedter to use ACI() less,

> Number of Vecursions
> test 1 test 2 test 1 test 2 simple back track 231 7969 6ms 25ms back Nach wish Binary forward 219 7782 & ms 69 ms back track with abnormal 202 2582 9 ms 42ms back with forward and MRV+LCV 70 1021 6 ms 286ms track checking back track with Arc Gusistancy 119 2178 22ms 1086 ms

