Modeling:

In Code, "not_initialized" is in domain of every variable and at first (before any assignment) all variables are "not_initialized".

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

Pair Consistency: For every X_{ij} → if $|X_{ij}|$ = Positive then Pair(X_{ij}) must be negative
→ if $|X_{ij}|$ = negative then Pair(X_{ij}) must be positive
→ if $|X_{ij}|$ = empty then pair(X_{ij}) must be empty

Pair: $\boxed{-\ +}$

X_{ij} Pair(X_{ij})

Pole consistency: for every X_{ij} → if $|X_{ij}|$ = positive, then $\{X_{ij+1}, X_{i+1j}, X_{ij-1}, X_{i-1j}\}$ can't be positive
→ if $|X_{ij}|$ = negative, then $\{X_{ij+1}, X_{i+1j}, X_{ij-1}, X_{i-1j}\}$ can't be negative

Row consistency:

for each Column j: $\sum_{i=1}^{n}$ Positive value of X_{ij} = row_positive[j]

for each Column j: $\sum_{i=1}^{n}$ Positive value of X_{ij} = row_negative[j]

Column Consistency:

for each row i : $\sum_{j=1}^{n}$ Positive value of X_{ij} = Col_positive[i]

for each row i : $\sum_{j=1}^{n}$ Positive value of X_{ij} = Col_negative[i]

Implementing Backtracking:

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 ← 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 ← RRECURSIVE-BACTRACKING(*assignment*, *csp*)
      if *result* ≠ *failure* **then return** *result*
    remove *{var=value}* from *assignment*
  return *failure*

```java
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");
        csp.printBoard( x: -1,  y: -1);
        return true;
    }
    //selection is based on index
    Variable varToAssign = null;
    for(int i = 0; i < csp.n; i++) {
        boolean stop = false;
        for(int j = 0; j < csp.m; j++) {
            if(csp.vars[i][j].value == VarState.notInit) {
                stop = true;
                varToAssign = csp.vars[i][j];
                break;
            }
        }
        if(stop) break;
    }
    for(VarState val : varToAssign.domain) {
        //assign
        csp.vars[varToAssign.row][varToAssign.col].value = val;
        if(csp.vars[varToAssign.row][varToAssign.col].isConsistent(csp)) {
            boolean result = recursive(csp);
            if(result) return true;
        }
        //remove from assignment
        csp.vars[varToAssign.row][varToAssign.col].value = VarState.notInit;
    }
    return false;
}
```

selecting variable to assign is based on Index

selecting value to assign is arbitrary

Implementing ForwardChecking:

- Source book mentions that Forward Checking only checks Binary Constraints, but in this problem, checking higher consistency is not very costly, so I wrote 3 forward checking functions:

  - Binary Forward Checking: only checks binary constraints (pair consistency + pole consistency)

```java
public static ArrayList<Pair> BinaryForwardChecking(Csp csp, Variable newAssignedVar) {
    ArrayList<Pair> oldDomains = new ArrayList<>();

    //-----------------------------------pairVar value consistency:
    pairForwardChecking(csp, newAssignedVar, oldDomains);

    //-----------------------------------pole consistency (up, down, right and left)
    poleForwardChecking(csp, newAssignedVar, oldDomains);

    return oldDomains;
}
```

all forward checking functions return List of previous domain of variables that their domain's changes, because we may need to restore previous domain when we back track.

```java
public static void pairForwardChecking(Csp csp, Variable newAssignedVar, ArrayList<Pair> oldDomains) {
    Variable pairVar = csp.getPair(newAssignedVar);
    oldDomains.add(new Pair(pairVar, (TreeSet<VarState>) pairVar.domain.clone()));
    //if it is unassigned:
    if(pairVar.value==VarState.notInit) {
        ArrayList<VarState> listToRemove = new ArrayList<>();
        for(VarState var : pairVar.domain) {
            //assign
            pairVar.value = var;

            //check pairVar con
            //// only check pair consistency, because we are checking binary constraint between 2 vars
            if(!pairVar.isPairConsistent(csp)) {
                //pairVar.domain.remove(var);
                listToRemove.add(var);
            }
            //undo assignment
            pairVar.value = VarState.notInit;
        }
        pairVar.domain.removeAll(listToRemove);
    }
}
public static void poleForwardChecking(Csp csp, Variable newAssignedVar, ArrayList<Pair> oldDomains) {
    for(int x = -1; x <= 1; x++) {
        for(int y = -1; y <= 1; y++) {
            if((x==0 && y==0) || x*y!=0)  continue;
            if(newAssignedVar.row+x < csp.n && newAssignedVar.row+x >= 0
                    && newAssignedVar.col+y < csp.m && newAssignedVar.col+y >= 0) {

                int i = newAssignedVar.row+x;
                int j = newAssignedVar.col+y;

                //update domain
                if(csp.vars[i][j].value==VarState.notInit) {

                    boolean alreadyAdded = false;
                    for(Pair p : oldDomains) if(p.var.row==i && p.var.col==j) alreadyAdded = true;
                    if(!alreadyAdded) oldDomains.add(new Pair(csp.vars[i][j], (TreeSet<VarState>) csp.vars[i][j].domain.clone()));

                    if(newAssignedVar.value==VarState.pos) {
                        //delete pos from i j's domain
                        if(csp.vars[i][j].domain.contains(VarState.pos)) csp.vars[i][j].domain.remove(VarState.pos);
                    }
                    else if(newAssignedVar.value==VarState.neg) {
                        if(csp.vars[i][j].domain.contains(VarState.neg)) csp.vars[i][j].domain.remove(VarState.neg);
                    }
                }
            }
        }
    }
}
```

  - ForwardChecking(): same implementation, except it also checks row/column consistency

  - abnormalForwardChecking(): same implementation, it can also removes "empty" from domain

Implementing MRV, LCV:

LCV: using Forward Checking, update domains with respect to assigning a varState, then counts the "flexibility" and goes to next var state, at the end returns sorted list of legal moves of the variable we choose to assign

(first varstate is the least constraining value)

```java
// returns list of legal values sorted by LCV ( ONLY CONSISTENT ONES )
public static ArrayList<VarState> LCV(Csp csp, Variable assignedVar) {
    HashMap<VarState, Integer> flexMap = new HashMap<>();
    ArrayList<VarState> sortedDomain = new ArrayList<>();

    for(VarState val : assignedVar.domain) {
        //assign
        int domainSum = 0;
        csp.vars[assignedVar.row][assignedVar.col].value = val;

        ArrayList<Pair> oldDomains = Csp.abnormalForwardChecking(csp, assignedVar);
        for(int i = 0; i < csp.n; i++) {
            for(int j = 0; j < csp.m; j++) {
                if(csp.vars[i][j].value == VarState.notInit) {
                    domainSum += csp.vars[i][j].domain.size();
                }
            }
        }
        //UNDO FORWARD CHECKING
        for(Pair p : oldDomains) {
            csp.vars[p.var.row][p.var.col].domain = p.domain;
        }

        flexMap.put(val, domainSum);
        sortedDomain.add(val);
    }


    sortedDomain.sort(new Comparator<VarState>() {
        @Override
        public int compare(VarState o1, VarState o2) {
            if(flexMap.get(o1) > flexMap.get(o2)) return -1;
            else if(flexMap.get(o1) < flexMap.get(o2)) return 1;
            else return 0;
        }
    });

    if(sortedDomain.size()==3 &&
            (flexMap.get(VarState.neg).intValue() != flexMap.get(VarState.pos).intValue())
            &&
            (flexMap.get(VarState.neg).intValue() != (flexMap.get(VarState.empty).intValue()))
            &&
            (flexMap.get(VarState.empty).intValue() != (flexMap.get(VarState.pos)).intValue())
    ){
        //System.out.println(flexMap);
        //System.out.println(sortedDomain);

    }
    return sortedDomain;
}
```

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

```java
//ADD -> select a var based on MRV
Variable varToAssign = csp.vars[0][0];
boolean firstNotInitializedFound = false;
for(int i = 0; i < csp.n; i++) {
    for(int j = 0; j < csp.m; j++) {
        if(csp.vars[i][j].value == VarState.notInit) {
            if(!firstNotInitializedFound) {
                varToAssign = csp.vars[i][j];
                firstNotInitializedFound = true;
            }
            else {
                if(csp.vars[i][j].domain.size() < varToAssign.domain.size()) {
                    varToAssign = csp.vars[i][j];
                }
            }
        }
    }
}
```

note that forward checking already updated domains up to this point.

effectiveness of MRV :

- In this problem, using MRV has <u>negative</u> effect compard to chosing variables based on their index. when choosing variable to assign based on their index (from left top to right bottom) we iterate over cells of one row and then go to the next row, meaning that when backtrack is in the second row, it guarantees that Row pos/neg constraints on first row are satisfied. (it makes sure that it satisfies current row constraints before moving to next row).
But using MRV, we might choose a variable in i-th row, and next variable that it chooses to assign is in (i+k)-th column.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

order of assignment in Index-based

1			3	
	2			
				5
		4		

order of assignment in MRV

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 csp

  while $queue$ 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** *false*
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add (X_k, X_i) to $queue$
  return *true*

function 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 functions:

- Arc-consistency only deals with Binary Constraints and does not check Row/Col Constraints because they are not binary, so it's not that strong.
- ac3() function has a large overhead and it's very COSTLY.
- although Arc Consistency is Stronger than Binary Forward Checking, it does not run faster,

for example for test2.txt

	with ac3()	with binaryForwardChecking()
number of recursions:	2178	7969
run time:	798ms	59 ms

- its obvious that Arc-Consisteny CAN be weaker than abnormal Forward Checking because it does not check Row/Col consistency
- it's probably better to use AC3() less, (for example every time 10 new variables gets assigned)

	number of recursions		exec time	
	test1	test2	test1	test2
simple backtrack	231	7969	6ms	25ms
back track with Binary forward checking	219	7782	8ms	69ms
back track with abnormal forward checking	202	2582	9ms	42ms
back track with abnormal forward checking and MRV+LCV	70	1021	6ms	286ms
back track with Arc Consistency	119	2178	22ms	1086 ms