EECS 3401 — AI and Logic Prog. — Lecture 14

Adapted from slides of Yves Lesperance

Vitaliy Batusov
vbatusov@cse.yorku.ca

York University

November 9, 2020

- Today: **Constraint Satisfaction**
- Required reading: Russell & Norvig Chapter 6

CSP, Formally

A **Constraint Satisfaction Problem** (CSP) consists of:

- A set of variables $V_1, \ldots, V_n$
- For each variable V_i, a domain of possible values $Dom[V_i]$
- A set of constraints $C_1, \ldots, C_m$

- Each variable V_i can be assigned any value from $Dom[V_i]$
- Each constraint C has
  - A **scope**: a subset of the problem's variables which it concerns e.g., V_1, V_2, V_4
  - A boolean function that maps assignments to these variables to True/False, e.g.,
    $C(V_1 = a, V_2 = b, V_4 = c) = True$
    $C(V_1 = b, V_2 = c, V_4 = c) = False$

A **solution** to a CSP is an assignment of a value to all of the variables such that every constraint is satisfied.

Example: Sudoku

- Variables: $V_{1,1}, V_{1,2}, \ldots, V_{9,9}$
- Domains:
  $Dom[V_{i,j}] = \{1..9\}$ for empty cells,
  $Dom[V_{i,j}] = \{k\}$ for each cell pre-filled with some k
- Row constraints:
  $CR_1(V_{1,1}, V_{1,2}, \ldots, V_{1,9})$
  $CR_2(V_{2,1}, V_{2,2}, \ldots, V_{2,9})$
  ...
  $CR_9(V_{9,1}, V_{9,2}, \ldots, V_{9,9})$
- Column constraints:
  $CC_1(V_{1,1}, V_{2,1}, \ldots, V_{9,1})$
  etc.
- Sub-square constraints:
  $CSS_1(V_{1,1}, V_{1,2}, V_{1,3}, V_{2,1}, V_{2,2}, V_{2,3}, V_{3,1}, V_{3,2}, V_{3,3})$
  etc.

Example: Sudoku

- Each of these constraints is over 9 variables, and they are all the same constraint: all values must be unique
- Such constraints are often called the `ALL-DIFF` constraints
- Thus, Sudoku has 3 x 9 `ALL-DIFF` constraints, one over each set of variables in the same column, one over each set of variables in the same row, and one over each set of variables in the same sub-square
- Note: An `ALL-DIFF` constraint over k variables can be equivalently expressed by k-choose-2 not-equal constraints over each pair of these variables.
  Let NEQ be a not-equal constraint; then
  $CSS_1(V_{1,1}, V_{1,2}, V_{1,3}, V_{2,1}, V_{2,2}, V_{2,3}, V_{3,1}, V_{3,2}, V_{3,3}) =$
  $NEQ(V_{1,1}, V_{1,2}), NEQ(V_{1,1}, V_{1,3}), \ldots, NEQ(V_{3,2}, V_{3,3})$

Example: Exam Scheduling

Constraints:

- For all pairs of finals i, j such that there is a student taking both:

$$NEQ(T_i, T_j)$$

- For all pairs of finals i, j:

$$C(T_i, T_j, S_i, S_j),$$

  - satisfied by any set of assignments in which $T_i \neq T_j$ or $S_i \neq S_j$
  - falsified by any set of assignments in which $T_i = T_j$ as well as $S_i = S_j$

Solving CSPs

- CSPs can be solved by a specialized version of the depth-first search
- Key intuitions:
  - We can build up to a solution by searching through the space of partial assignments
  - Order in which we assign the variables does not matter—eventually, they all have to be assigned
  - If, during the process of building up a solution, we falsify a constraint, we can immediately reject all possible ways of extending the current partial assignment

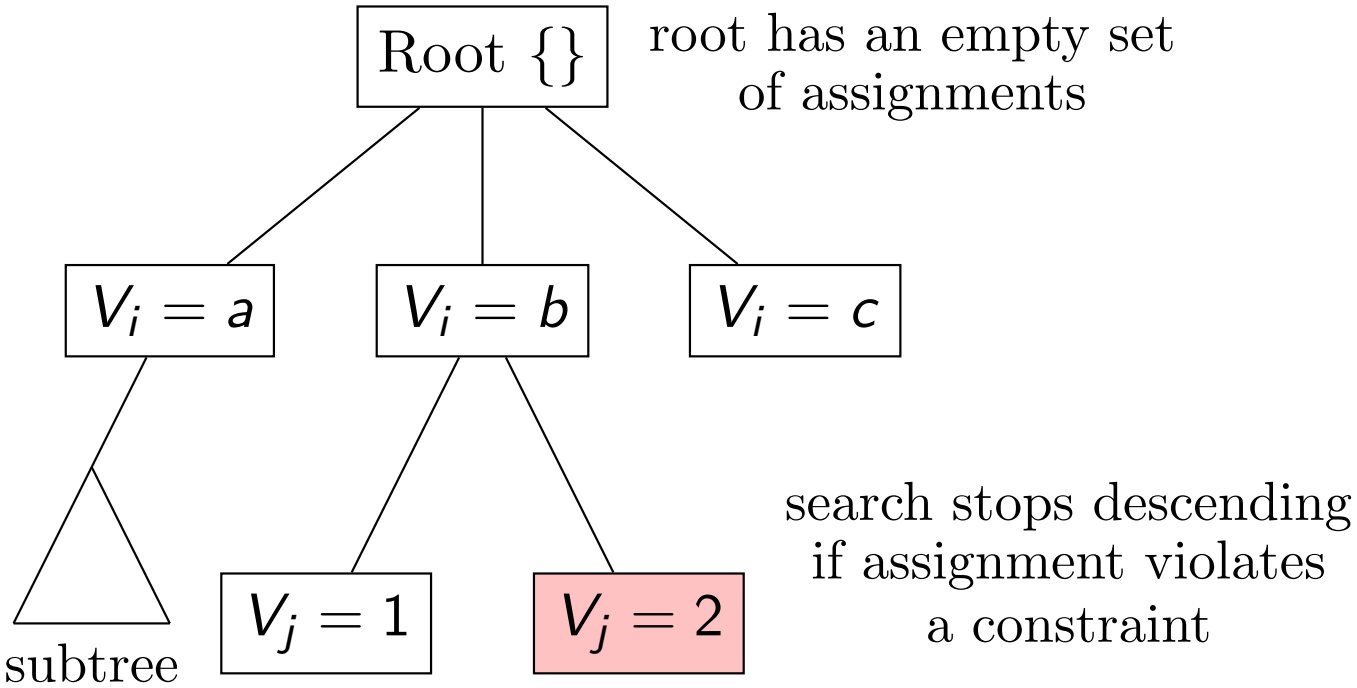
Backtracking Search

The following backtracking search algorithm is based on these ideas:

```
BT(Level):
    If all variables assigned:
        return all values
    V := PickUnassignedVariable()
    Variable[Level] := V
    Assigned[V] := true
    for each member d of Domain(V):
        Value[V] := d
        OK := TRUE
        for each constraint C such that V is a variable of C
                        and all other variables of C are assigned:
            if C is not satisfied by the current set of assignments:
                OK := FALSE
    if(OK):
        BT(Level+1)
return
```

Solving CSPs

The algorithm searches a tree of partial assignments



Backtracking Search

- Heuristics are used to determine which variable to assign next

  In pseudocode above, `PickUnassignedVariable()`
- The choice can vary from branch to branch

  For example, under the assignment $V_1 = a$, we might choose to assign V_4 next, while under $V_1 = b$, we might choose to assign V_5 next
- This *dynamically*-chosen variable ordering has a tremendous impact on performance

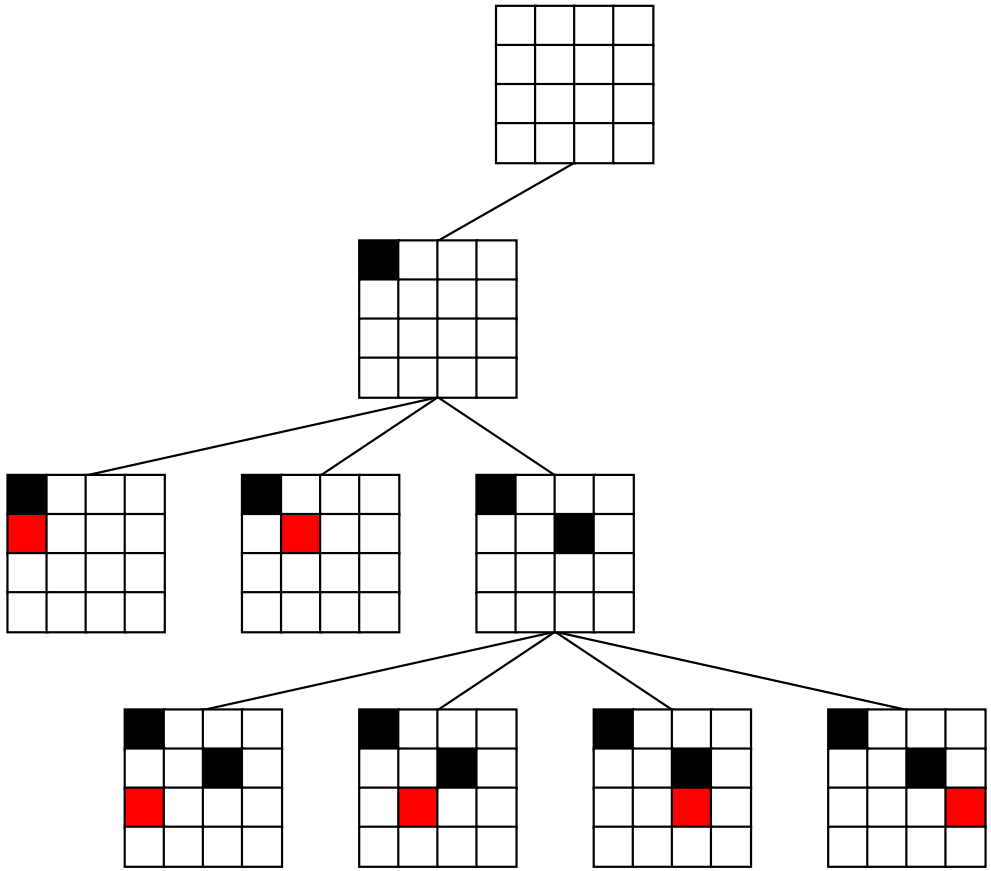
Example

The N-Queens Problem:
Place N Queens on an $N \times N$ chess board so that no Queen can attack any other Queen

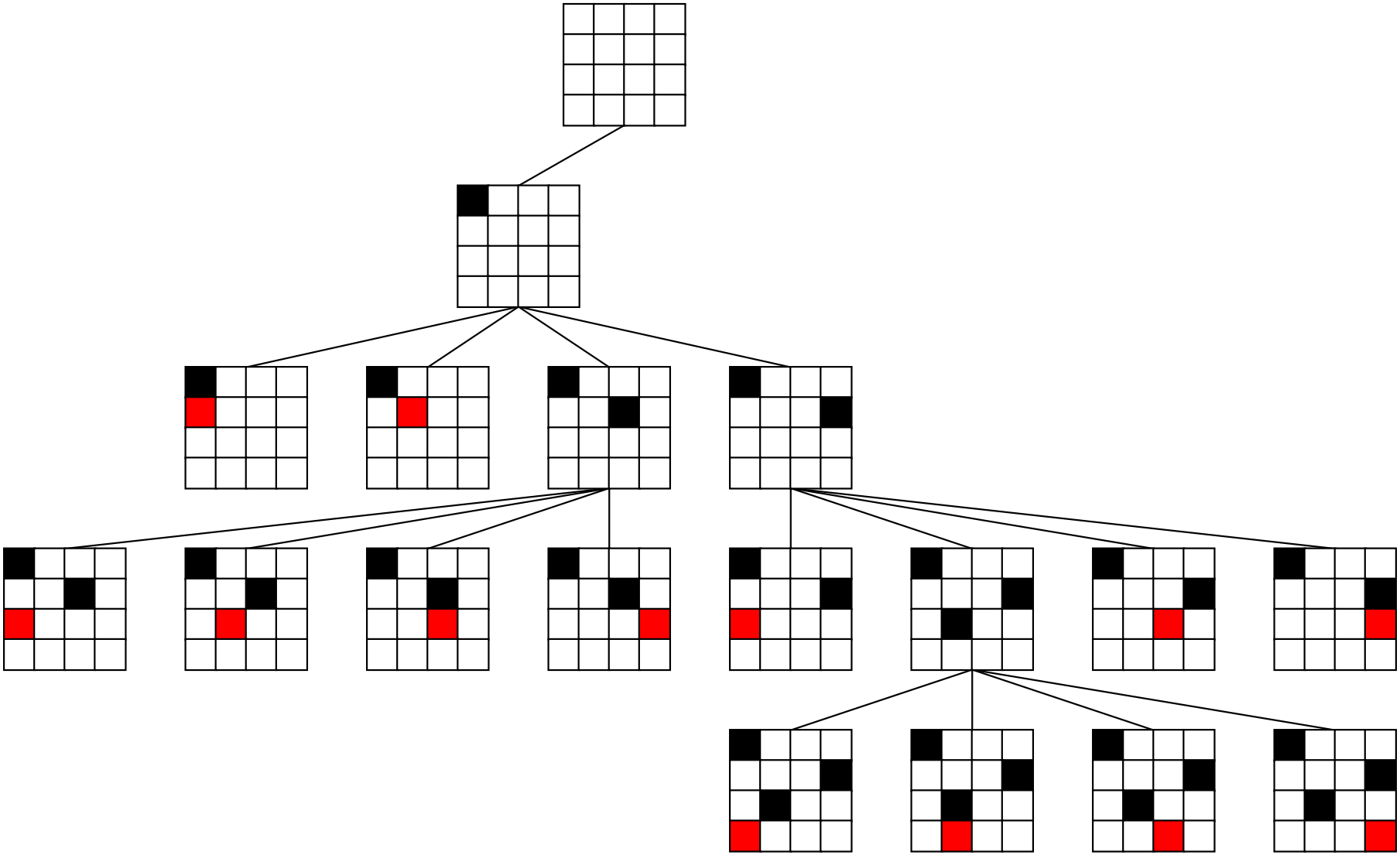
- Variables: V_i $(1 \leq i \leq N)$, one per row (why?)
  Value of V_i defines the column on row i where a Queen is placed

- Constraints:
  - $V_i \neq V_j$ for all $i \neq j$ (can't put two Queens on same column)
  - $|V_i - V_j| \neq i - j$ (diagonal constraint)

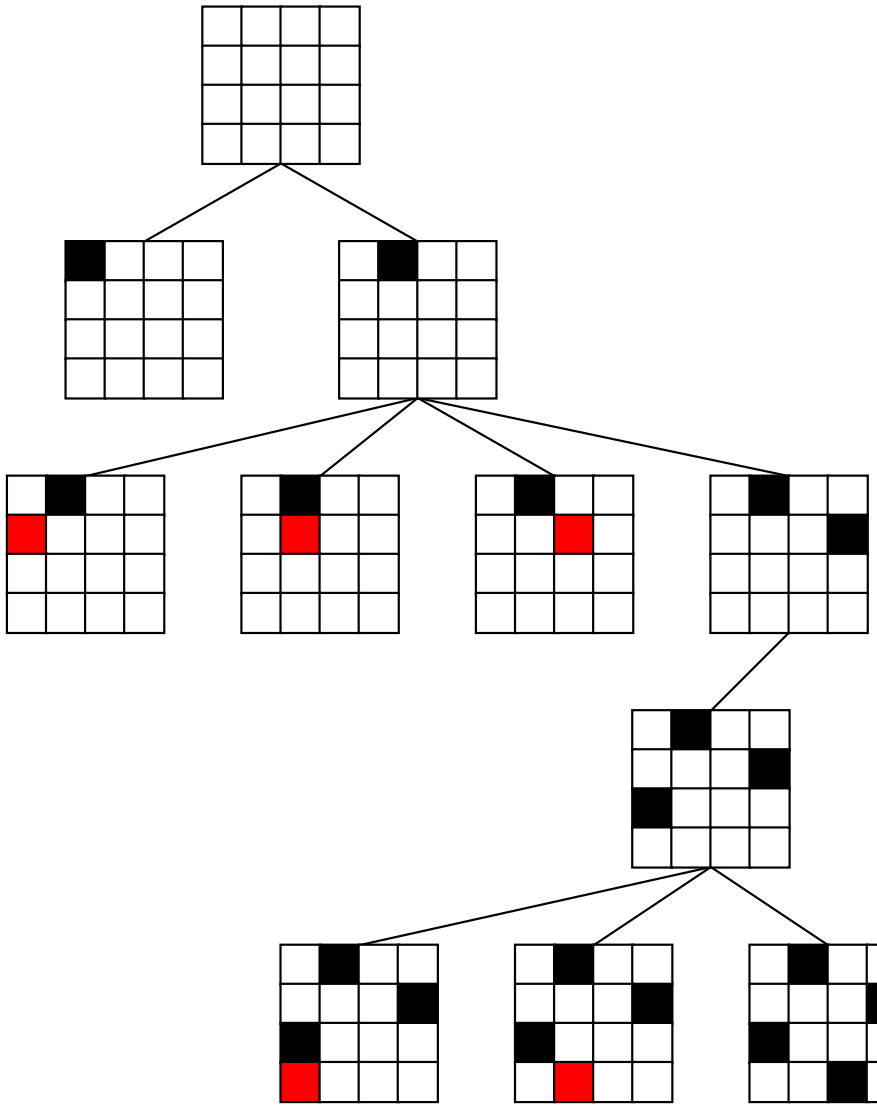
Example: 4×4 Queens



Example: 4×4 Queens



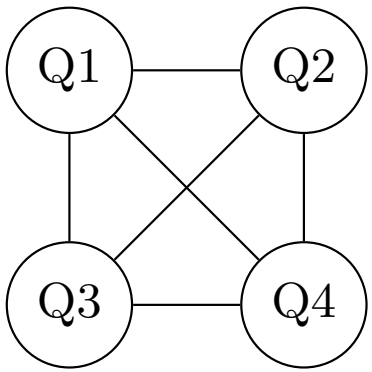
Example: 4×4 Queens



Solution!

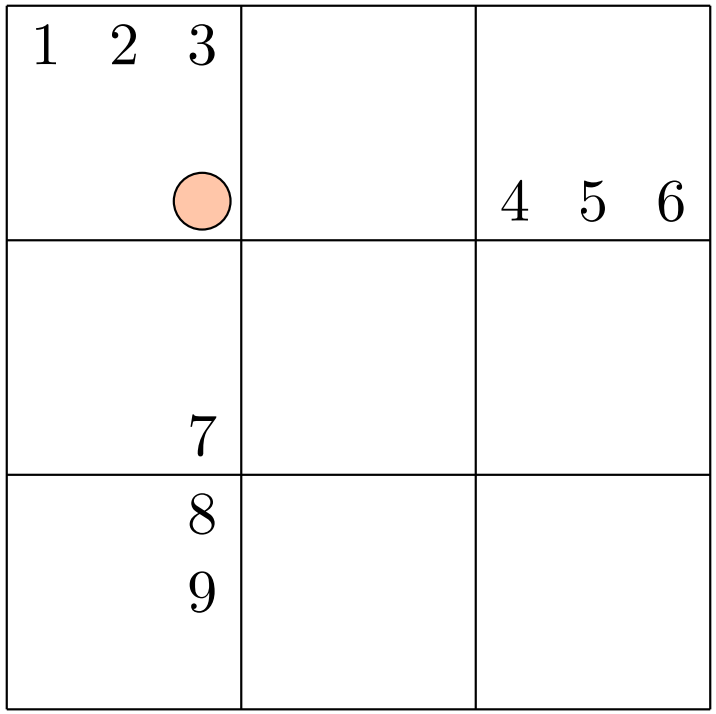
Backtracking Search

- **Unary Constraints**: over one variable
  $C(X) : X = 2$, $C(Y) : Y > 5$
- **Binary Constraints**: over two variables
  $C(X, Y) : X + Y < 6$
  Can be represented by a **constraint graph**, where nodes are variables and arcs are the constraints. E.g., 4-Queens:



- **Higher-Order Constraints**: over 3 or more variables
  Can convert any constraint into a set of binary constraints (may need some auxiliary variables)

Problems with plain backtracking



The cell 3,3 has no possible value. But in the backtracking search we don't detect this until all variables of a row/column/sub-square constraint are assigned

Constraint Propagation

- Constraint propagation refers to the technique of *looking ahead* in the search at the as-yet unassigned variables
- Try to detect if any "obvious" failures have occurred
- "Obvious" refers to things we can test/detect efficiently
- Even if we don't detect an obvious failure, we might be able to eliminate some possible part of the future search

Constraint Propagation

- Propagation has to be applied *during* search, potentially at every node of the search tree
- If propagation is slow, this can slow the search down to the point where using propagation actually slows search down!
- There is always a trade-off between searching fewer nodes in the search and having a higher node-per-second processing rate

Forward Checking

- Forward checking is an extension of backtracking search that employs a "modest" amount of propagation (lookahead)
- When a variable is instantiated, we check all constraints that have **only one uninstantiated variable** remaining
- For that uninstantiated variable, we check all of its values, pruning those values that violate the constraint

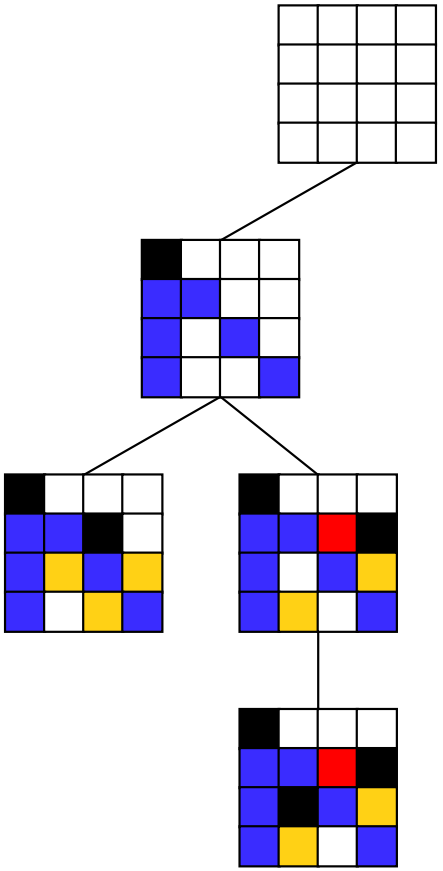
Forward Checking

```
FCCheck(C, x):   // C is constraint, x its only uninst. var.
    for d := each member of CurDom[x]
        if making x = d together with previous assignments
            to variables in scope C falsifies C:
        remove d from CurDom[V]
    if CurDom[V] = {} then return DWO (Domain Wipe Out)
return ok
```

Forward Checking

```
FC(Level):
If all variables are assigned:
    return Value of each Variable // as before
    V := PickAnUnassignedVariable()
    Variable[Level] := V
    Assigned[V] := TRUE
    for d := each member of CurDom(V)
        Value[V] := d
        for each constraint C over V that has one
                unassigned variable in its scope X:
            val := FCCheck(C,X)
            if(val != DWO)
                FC(Level+1)
        RestoreAllValuesPrunedByFCCheck()
return
```

Forward Checking Example: 4-Queens



Restoring Values

- After we backtrack from current assignment (in the for-loop), we must restore the values that were pruned as a result of that assignment
- Some bookkeeping needs to be done, as we must remember which values were pruned by which assignment
- FC also gives for free a very powerful heuristic:
  - Always branch on a variable with the smallest remaining values (smallest CurDom)
  - If a variable has only one value left, that value is forced, so we should propagate its consequences immediately
  - This heuristic tends to produce skinny trees at the top. This means that more variables can be instantiated with fewer nodes searched, and thus more constraint propagation/DWO failures occur with less work

Empirically

- FC often about 100 faster than BT
- FC with MRV (minimum remaining values) often 10000 times faster
- But on some problems the speed up can be much greater

  Converts problems that are essentially not solvable to problems that are solvable

End of Lecture

- Next time: **Planning as Search**