Constraint Satisfaction Problems (Backtracking Search)

- Chapter 6
  - 6.1: Formalism
  - 6.2: Constraint Propagation
  - 6.3: Backtracking Search for CSP
  - 6.4 is about local search which is a very useful idea but we won't cover it in class.

Constraint Satisfaction Problems

- As we’ve discussed search problems, we’ve designed problem specific state representations for our search routines to use.
- We’ve also generally been concerned not only to arrive at goal states, but to determine paths for our agents to follow.
- We might, however, care less about paths and more about final goal states or configurations.
- We might also prefer general state representations that can take advantage of specialized search algorithms.
- We call search problems that can be formalized in this specialized way **CSP**s, or Constraint Satisfaction problems.
- CPSs are search problems with a uniform, simple state representation that allows design of more efficient algorithms.
- Techniques for solving CSPs have many practical applications in industry.

Constraint Satisfaction Problems

State Representation:

Factored into variables that can take on values

Algorithms:

General purpose, enforce constraints on factors

Main Idea:

Eliminate chances of search space by identifying value assignments for variables that **violate constraints**.

Representing States with Feature Vectors

- For each problem we have designed a new state representation (and designed the sub-routines called by search based on this representation).
- **Feature vectors** provide a general state representation that is useful for many different problems.
- Feature vectors are also used in many other areas of AI, particularly Machine Learning, Reasoning under Uncertainty, Computer Vision, etc.

Feature Vectors

- **State Representation:** states are vectors of feature values

- We have
  - A set of k **features** (or **variables**)
  - Each variable has a **domain** of different values.
  - A **state** is specified by an assignment of a value for each variable.
    - height = {short, average, tall},
    - weight = {light, average, heavy}
  - A **partial state** is specified by an assignment of a value to some of the variables.

- In CSPs, the problem is to search for a set of values for the features (variables) so that the values satisfy some conditions (constraints).
  - I.e., a goal state specified as conditions on the vector of features values.

Example: Sudoku

	2							
			6					3
	7	4		8				
					3			2
	8			4			1	
6			5					
				1		7	8	
5					9			
							4	

1	2	6	4	3	7	9	5	8
8	9	5	6	2	1	4	7	3
3	7	4	9	8	5	1	2	6
4	5	7	1	9	3	8	6	2
9	8	3	2	4	6	5	1	7
6	1	2	5	7	8	3	9	4
2	6	9	3	1	4	7	8	5
5	4	8	7	6	9	2	3	1
7	3	1	8	5	2	6	4	9

Example: Sudoku

- 81 **variables**, each representing the value of a cell.
- **Domain of Values**: a single value for those cells that are already filled in, the set {1, …9} for those cells that are empty.
- **State:** any completed board given by specifying the value in each cell (1-9, or blank).
- **Partial State:** some incomplete filling out of the board.

Example: 8-Puzzle

2	**3**	**7**
6	4	8
5	1	

- **Variables:** 9 variables $\text{Cell}_{1,1}$, $\text{Cell}_{1,2}$, …, $\text{Cell}_{3,3}$
- **Values:** {'B', 1, 2, …, 8}
- **State:** Each "$\text{Cell}_{i,j}$" variable specifies what is in that position of the tile.
  - If we specify a value for each cell we have completely specified a state.

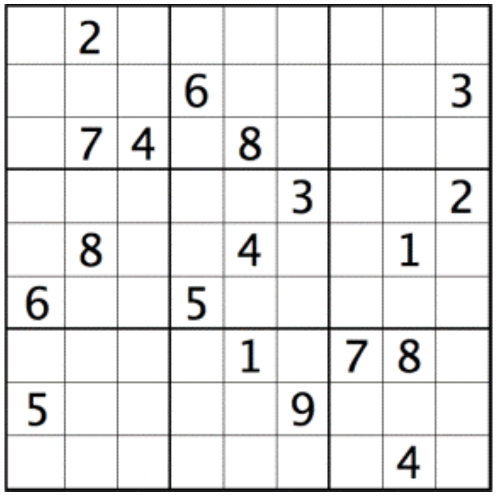
This is only one of many ways to specify the state.

Constraint Satisfaction Problems

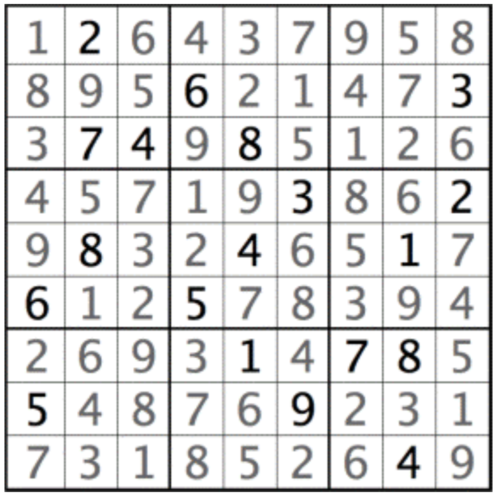
- Notice that in these problems some settings of the variables are **illegal.**
  - In Sudoku, we can't have the same number in any column, row, or subsquare.
  - In the 8 puzzle each variable must have a distinct value (same tile can't be in two places)

Constraint Satisfaction Problems

- In many practical problems finding which setting of the feature variables yields a legal state is difficult.



	2							
			6					3
	7	4		8				
					3			2
	8			4			1	
6			5					
				1		7	8	
5					9			
							4	

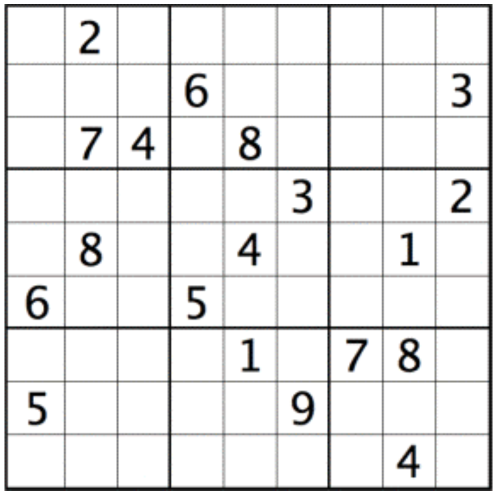


1	2	6	4	3	7	9	5	8
8	9	5	6	2	1	4	7	3
3	7	4	9	8	5	1	2	6
4	5	7	1	9	3	8	6	2
9	8	3	2	4	6	5	1	7
6	1	2	5	7	8	3	9	4
2	6	9	3	1	4	7	8	5
5	4	8	7	6	9	2	3	1
7	3	1	8	5	2	6	4	9

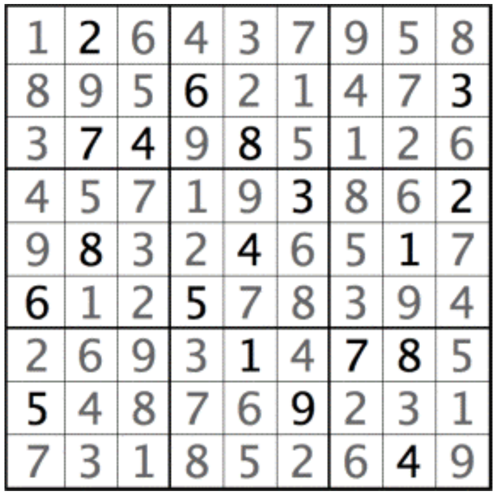
- We want to find a state (setting of the variables) that satisfies certain constraints.

Constraint Satisfaction Problems

- In Suduko: The variables that form
  - a column must be distinct
  - a row must be distinct
  - a sub-square must be distinct.



	2							
			6					3
	7	4		8				
					3			2
	8			4			1	
6			5					
				1		7	8	
5					9			
							4	



1	2	6	4	3	7	9	5	8
8	9	5	6	2	1	4	7	3
3	7	4	9	8	5	1	2	6
4	5	7	1	9	3	8	6	2
9	8	3	2	4	6	5	1	7
6	1	2	5	7	8	3	9	4
2	6	9	3	1	4	7	8	5
5	4	8	7	6	9	2	3	1
7	3	1	8	5	2	6	4	9

Constraint Satisfaction Problems

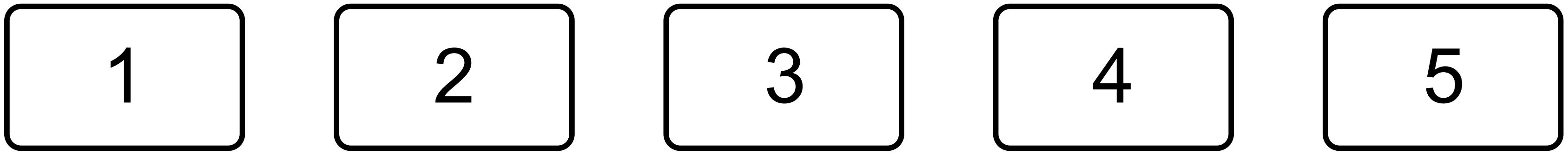
- In these problems we **do not care** about the sequence of moves needed to get to a goal state.
- We only care about finding a feature vector (a setting of the variables) that satisfies the goal.
  - A setting of the variables that satisfies some constraints.

- In contrast, in the 8-puzzle, the feature vector satisfying the goal is given. We care about the sequence of moves needed to move the tiles into that configuration

Example Car Sequencing

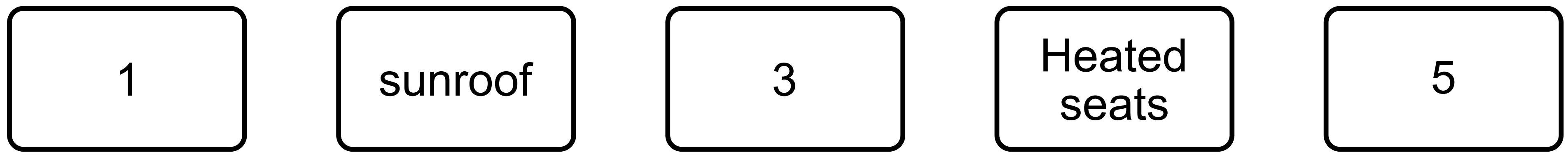
Car Factory Assembly Line—back to the days of Henry Ford

Move the items to be assembled don't move the workers



The assembly line is divided into stations. A particular task is preformed at each station.

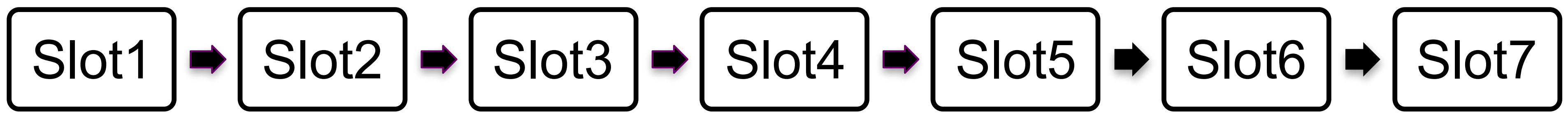
Example Car Sequencing



Some stations install optional items…not every car in the assembly line is worked on in that station.

As a result the factory is designed to have lower capacity in those stations.

Example Car Sequencing

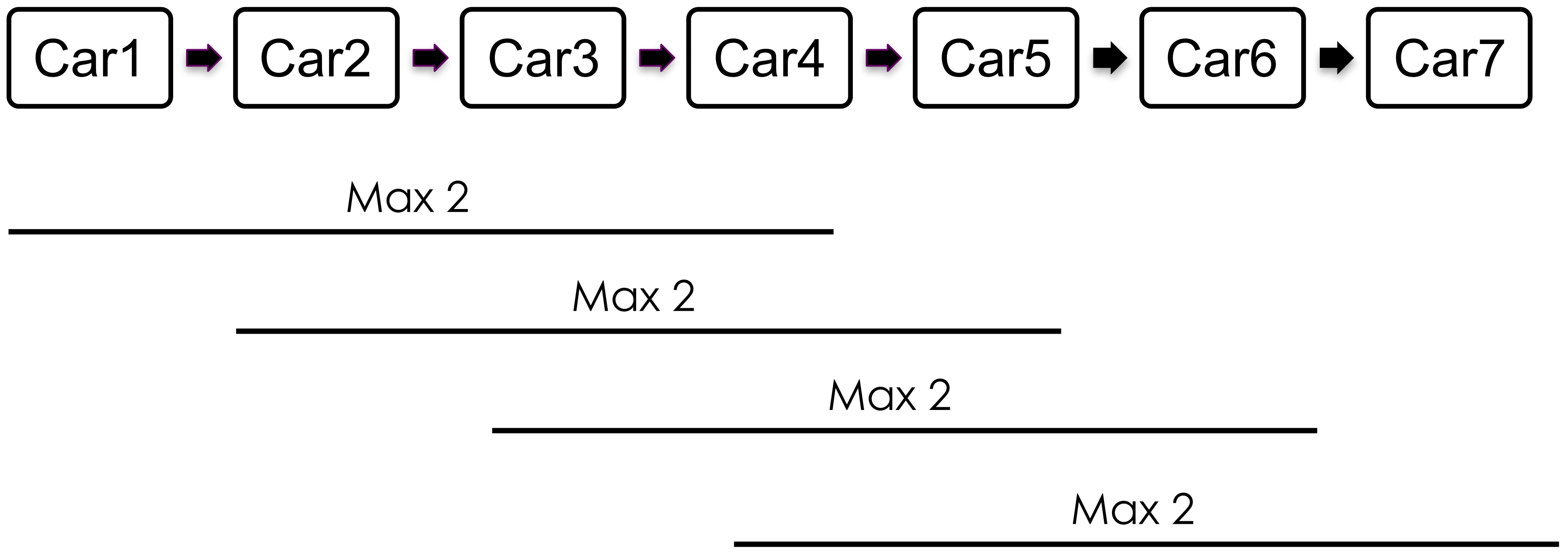


Cars move through the factory on an assembly line which is broken up into slots.

The stations might be able to process only a limited number of slots out of some group of slots that is passing through the station at any time.

E.g., the sunroof station might accommodate 4 slots, but only has capacity to process 2 slots out of the 4 at any one time.

Example Car Sequencing



Example Car Sequencing

$$\begin{tikzcd}\boxed{\text{Car1}} \arrow[r] & \boxed{\text{Car2}} \arrow[r] & \boxed{\text{Car3}} \arrow[r] & \boxed{\text{Car4}} \arrow[r] & \boxed{\text{Car5}} \arrow[r] & \boxed{\text{Car6}} \arrow[r] & \boxed{\text{Car7}}\end{tikzcd}$$

Each car to be assembled has a list of required options.

We want to assign each car to be assembled to a slot on the line.

But we want to ensure that no sequence of 4 slots has more than 2 cars assigned that require a sun roof.

Finding a feasible assignment of cars with different options to slots without violating the capacity constraints of the different stations is hard.

Formalization of a CSP

- A CSP consists of
  - A set of **variables** $V_1, \ldots, V_n$
  - For each variable a (finite) **domain** of possible values $Dom[V_i]$.
  - A set of **constraints** $C_1, \ldots, C_m$.

  - A **solution** to a CSP is an assignment of a value to all of the variables such that ***every constraint is satisfied***.
  - A CSP is unsatisfiable if no solution exists.

Formalization of a CSP

- Each variable can be assigned any value from its domain.
    - Vi = d where d ∈ Dom[Vi]

- Each constraint C
    - Has a set of variables it is over, called its **scope**
        - E.g., C(V1, V2, V4) is a constraint over the variables V1, V2, and V4. Its scope is {V1, V2, V4}
    - Given an assignment to its variables the constraint returns:
        - **True**—this assignment satisfies the constraint
        - **False**—this assignment falsifies the constraint.

Formalization of a CSP

- **Unary** Constraints (over one variable)
  - e.g. C(X):X=2; C(Y): Y>5
- **Binary** Constraints (over two variables)
  - e.g. C(X,Y): X+Y<6
- **Higher-order** constraints: over 3 or more variables.

Formalization of a CSP

- We can specify the constraint with a table
- C(V1, V2, V4) with Dom[V1] = {1,2,3} and Dom[V2] = Dom[V4] = {1, 2}

V1	**V2**	**V4**	**C(V1,V2,V4)**
1	1	1	False
1	1	2	False
1	2	1	False
1	2	2	False
2	**1**	**1**	**TRUE**
2	1	2	False
2	2	1	False
2	2	2	False
3	1	1	False
3	**1**	**2**	**TRUE**
3	**2**	**1**	**TRUE**
3	2	2	False

Formalization of a CSP

- Often we can specify the constraint more compactly with an expression: C(V1, V2, V4) = (V1 = V2+V4)

V1	V2	V4	C(V1,V2,V4)
1	1	1	False
1	1	2	False
1	2	1	False
1	2	2	False
2	1	1	**TRUE**
2	1	2	False
2	2	1	False
2	2	2	False
3	1	1	False
3	1	2	**TRUE**
3	2	1	**TRUE**
3	2	2	False

Example: Sudoku

- **Variables**: $V_{11}, V_{12}, \ldots, V_{21}, V_{22}, \ldots, V_{91}, \ldots, V_{99}$
- **Domains**:
  - Dom[V_{ij}] = {1-9} for empty cells
  - Dom[V_{ij}] = {k} a fixed value k for filled cells.
- **Constraints**:
  - Row constraints:
    - All-Diff($V_{11}, V_{12}, V_{13}, \ldots, V_{19}$)
    - All-Diff($V_{21}, V_{22}, V_{23}, \ldots, V_{29}$)
    - ...., All-Diff($V_{91}, V_{92}, \ldots, V_{99}$)
  - Column Constraints:
    - All-Diff($V_{11}, V_{21}, V_{31}, \ldots, V_{91}$)
    - All-Diff($V_{21}, V_{22}, V_{13}, \ldots, V_{92}$)
    - ...., All-Diff($V_{19}, V_{29}, \ldots, V_{99}$)
  - Sub-Square Constraints:
    - All-Diff($V_{11}, V_{12}, V_{13}, V_{21}, V_{22}, V_{23}, V_{31}, V_{32}, V_{33}$)
    - All-Diff($V_{14}, V_{15}, V_{16}, \ldots, V_{34}, V_{35}, V_{36}$)

Example: Sudoku

- Each of these constraints is over 9 variables, and they are all the same constraint:
  - Any assignment to these 9 variables such that each variable has a different value <u>satisfies</u> the constraint.
  - Any assignment where two or more variables have the same value <u>falsifies</u> the constraint.

- This is a special kind of constraint called an ALL-DIFF constraint.
  - ALL-Diff(V1, .., Vn) could also be encoded as a set of binary not-equal constraints between all possible pairs of variables: V1 ≠ V2, V1 ≠ V3, …, V2 ≠ V1, …, Vn ≠ V1, …, Vn ≠ Vn-1

Example: Sudoku

- Thus Sudoku has 3x9 ALL-DIFF constraints, one over each set of variables in the same row, one over each set of variables in the same column, and one over each set of variables in the same sub-square.

Solving CSPs

- Because CSPs do not require finding a paths (to a goal), it is best solved by a specialized version of 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. We can decide on a suitable value for one variable at a time!

  ➔ This is the key idea of backtracking search.
  - If we falsify a constraint during the process of building up a solution, we can immediately reject the current partial assignment:
    - All extensions of this partial assignment will falsify that constraint, and thus none can be solutions.

CSP as a Search Problem

A CSP could be viewed as a more traditional search problem

- Initial state: empty assignment
- Successor function: a value is assigned to any unassigned variable, which does not cause any constraint to return false.
- Goal test: the assignment is complete

Backtracking Search: The Algorithm BT

- In the slides that follow, we present a generic backtracking algorithm:
    - We pick a variable, assign it a value, test the constraints that we can. If a constraint is unsatisfied we **backtrack**, otherwise we set another variable. When all variables are set, we're done.
- Later in this unit, we refine this algorithm to include some constraint propagation (inference). More on this later!
- There are clever ways to pick what variable to assign and also what value to assign to it! Can you think of any? Think about what you do when you play Sudoku!

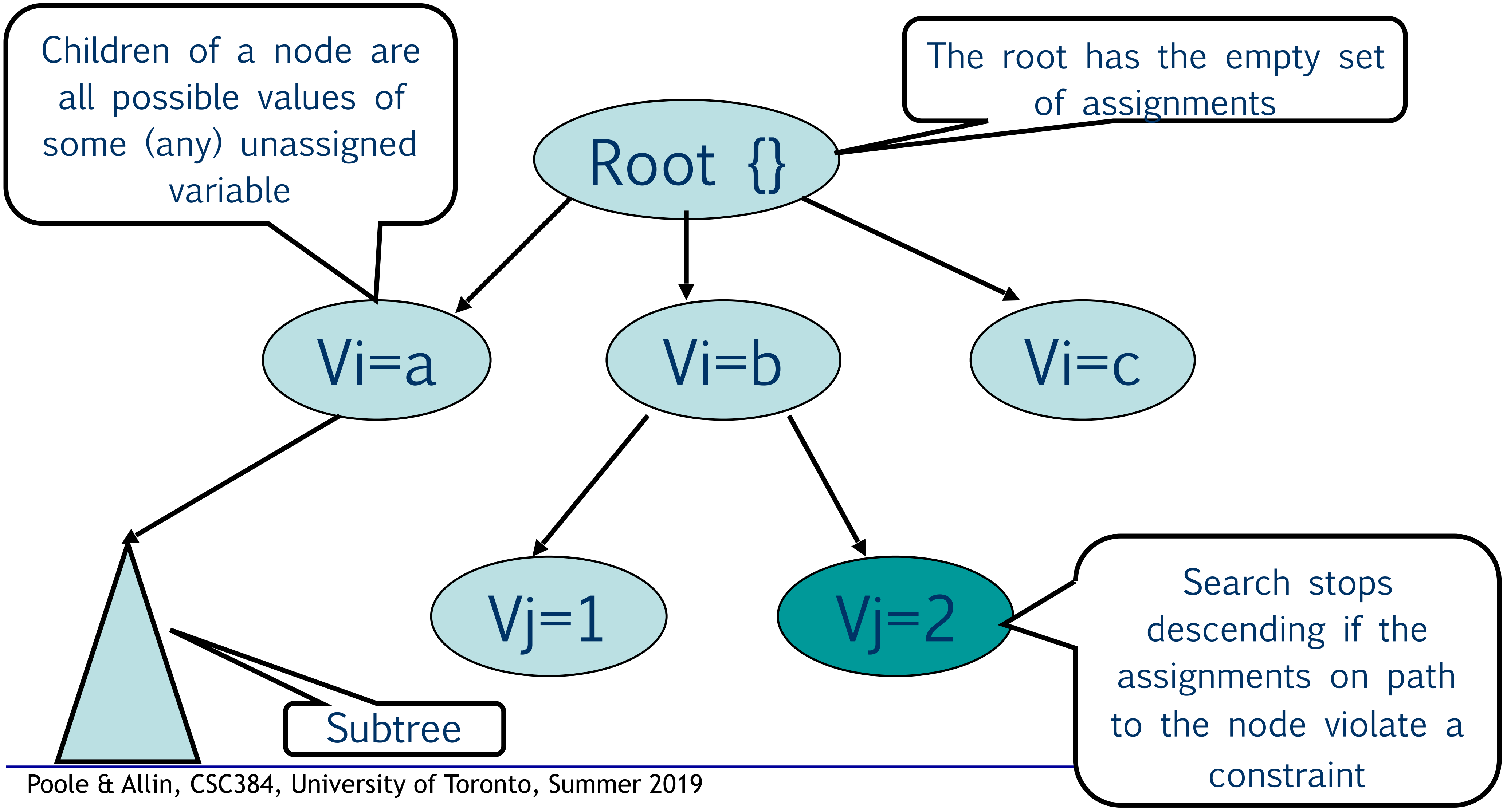
Backtracking Search: The Algorithm BT

```
BT(Level)
  If all variables assigned
       PRINT Value of each Variable
       RETURN or EXIT (RETURN for more solutions)
                       (EXIT for only one solution)
  V := PickUnassignedVariable()
  Assigned[V] := TRUE
  for d := each member of Domain(V) (the domain values of V)
       Value[V] := d
       ConstraintsOK = TRUE
       for each constraint C such that
              a) V is a variable of C and
              b) all other variables of C are assigned:
           IF C is not satisfied by the set of current
              assignments:
              ConstraintsOK = FALSE
       If ConstraintsOk == TRUE:
           BT(Level+1)

  Assigned[V] := FALSE //UNDO as we have tried all of V's values
  return
```

Backtracking Search

- The algorithm searches a tree of partial assignments.


Children of a node are all possible values of some (any) unassigned variable
The root has the empty set of assignments
Root {}
Vi=a
Vi=b
Vi=c
Vj=1
Vj=2
Subtree
Search stops descending if the assignments on path to the node violate a constraint

Backtracking Search

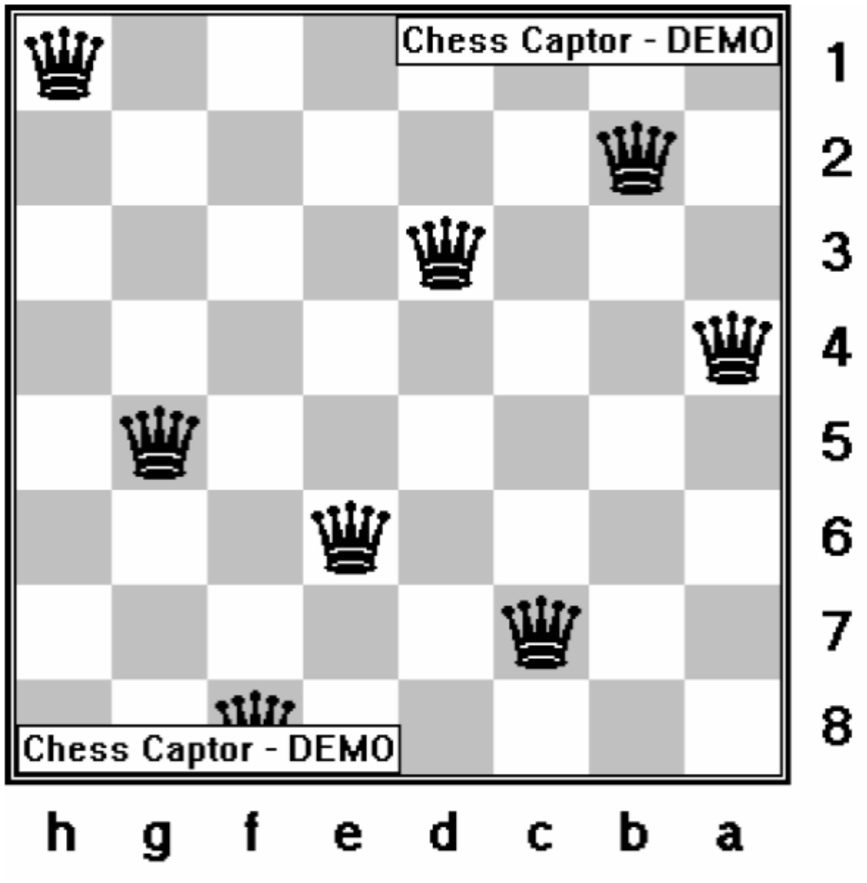
- Heuristics are used to determine
  - the order in which variables are assigned: `PickUnassignedVariable()`
  - the order of values tried for each variable.

- The choice of the next variable can vary from branch to branch, e.g.,
  - under the assignment V1=a we might choose to assign V4 next, while under V1=b we might choose to assign V5 next.

- This "dynamically" chosen variable ordering has a tremendous impact on performance.

Example: N-Queens

- Place N Queens on an N X N chess board so that no Queen can attack any other Queen.

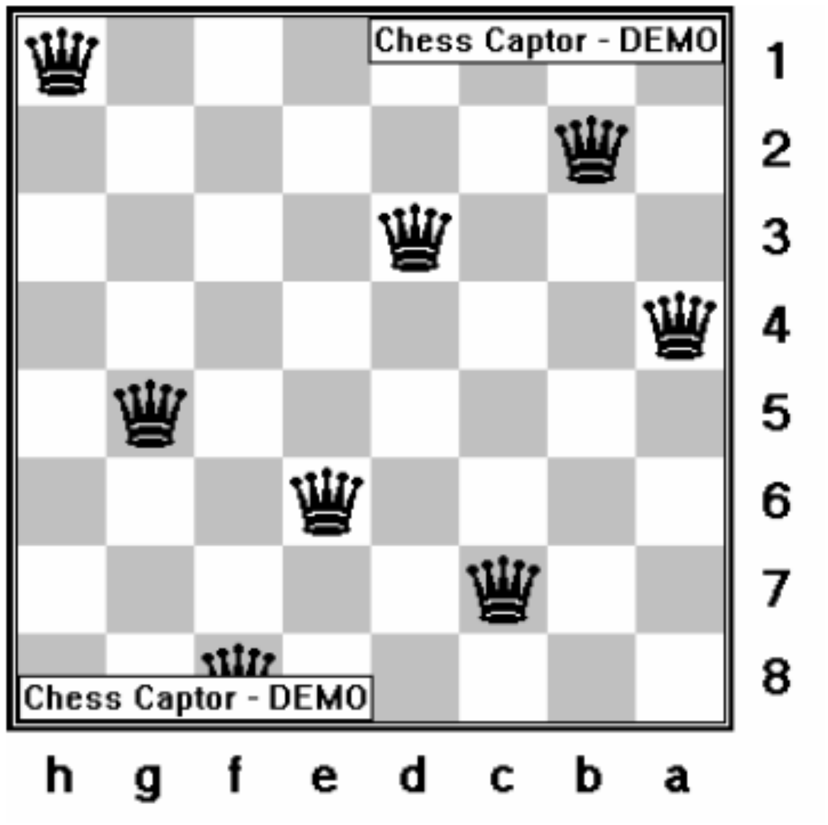


Example: N-Queens

- Problem formulation:
  - N variables (N queens)
  - N^2 values for each variable representing the positions on the chessboard
    - Value i is i'th cell counting from the top left as 1, going left to right, top to bottom.

Example: N-Queens

- Q1 = 1, Q2 = 15, Q3 = 21, Q4 = 32, Q5 = 34, Q6 = 44, Q7 = 54, Q8 = 59



Example: N-Queens

- This representation has $(N^2)^N$ states (different possible assignments in the search space)
  - For 8-Queens: 64^8 = 281,474,976,710,656

- Is there a better way to represent the N-queens problem?
  - We know we cannot place two queens in a single row → we can exploit this fact in the choice of the CSP representation

Example: N-Queens

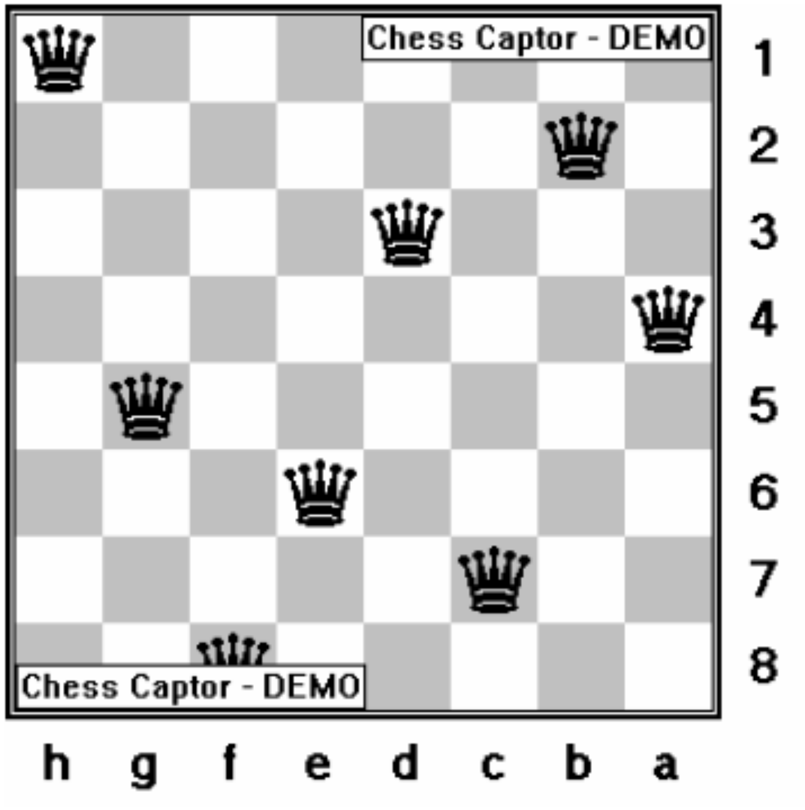
- Better Modeling:
  - N variables Qi, one per row.
  - Value of Qi is the column the Queen in row i is placed; possible values {1, …, N}.

- This representation has N^N states:
  - For 8-Queens: 8^8 = 16,777,216

- The choice of a representation can make the problem solvable or unsolvable!

Example: N-Queens

- Q1 = 1, Q2 = 7, Q3 = 5, Q4 = 8, Q5 = 2, Q6 = 4, Q7 = 6, Q8 = 3



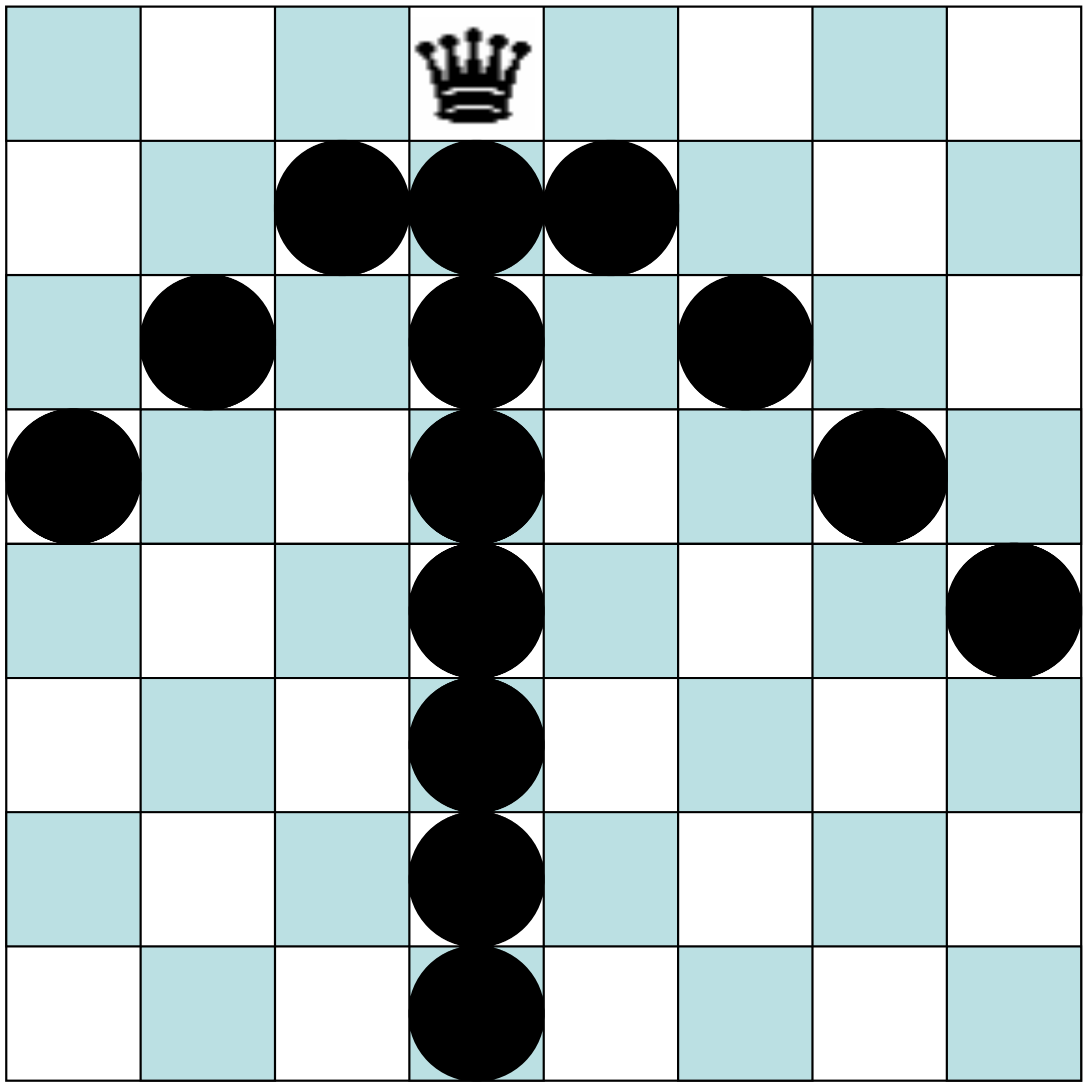
Example: N-Queens

- Constraints:
  - Can't put two Queens in same column

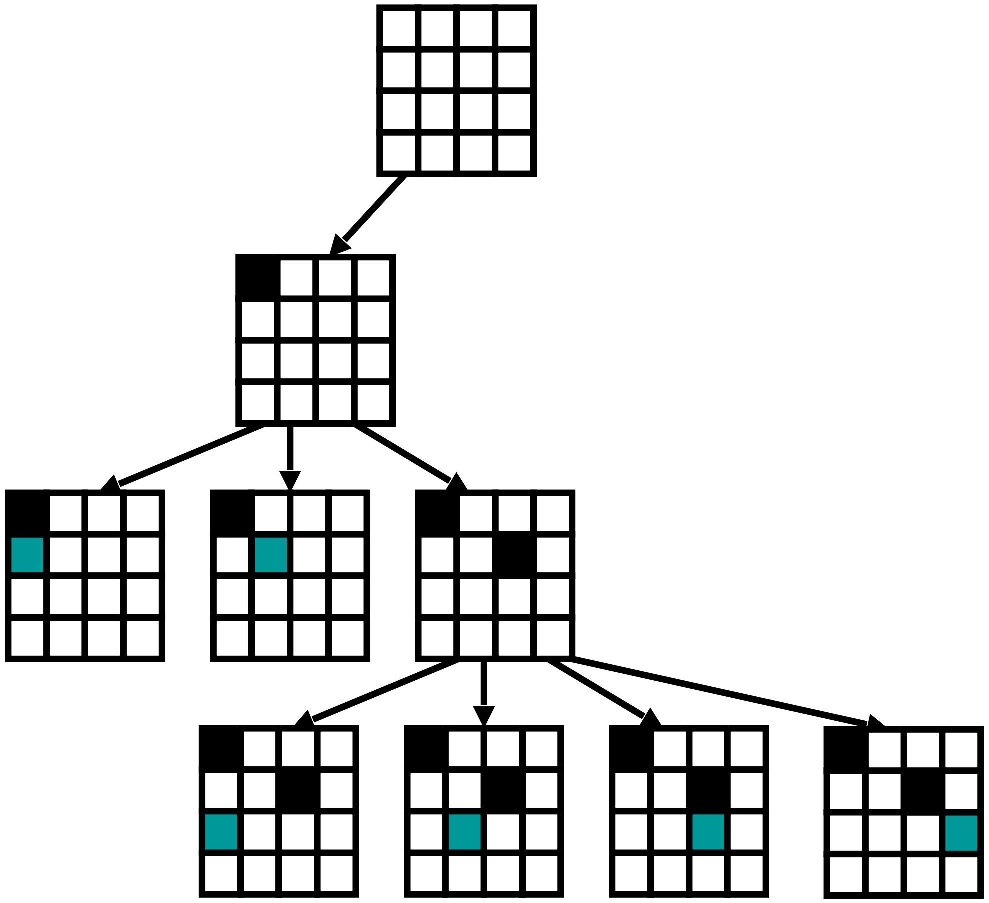
    Qi ≠ Qj for all i ≠ j
  - Diagonal constraints

    abs(Qi-Qj**)** ≠ **abs(**i-j**)**
    - i.e., the difference in the values assigned to Qi and Qj can't be equal to the difference between i and j.

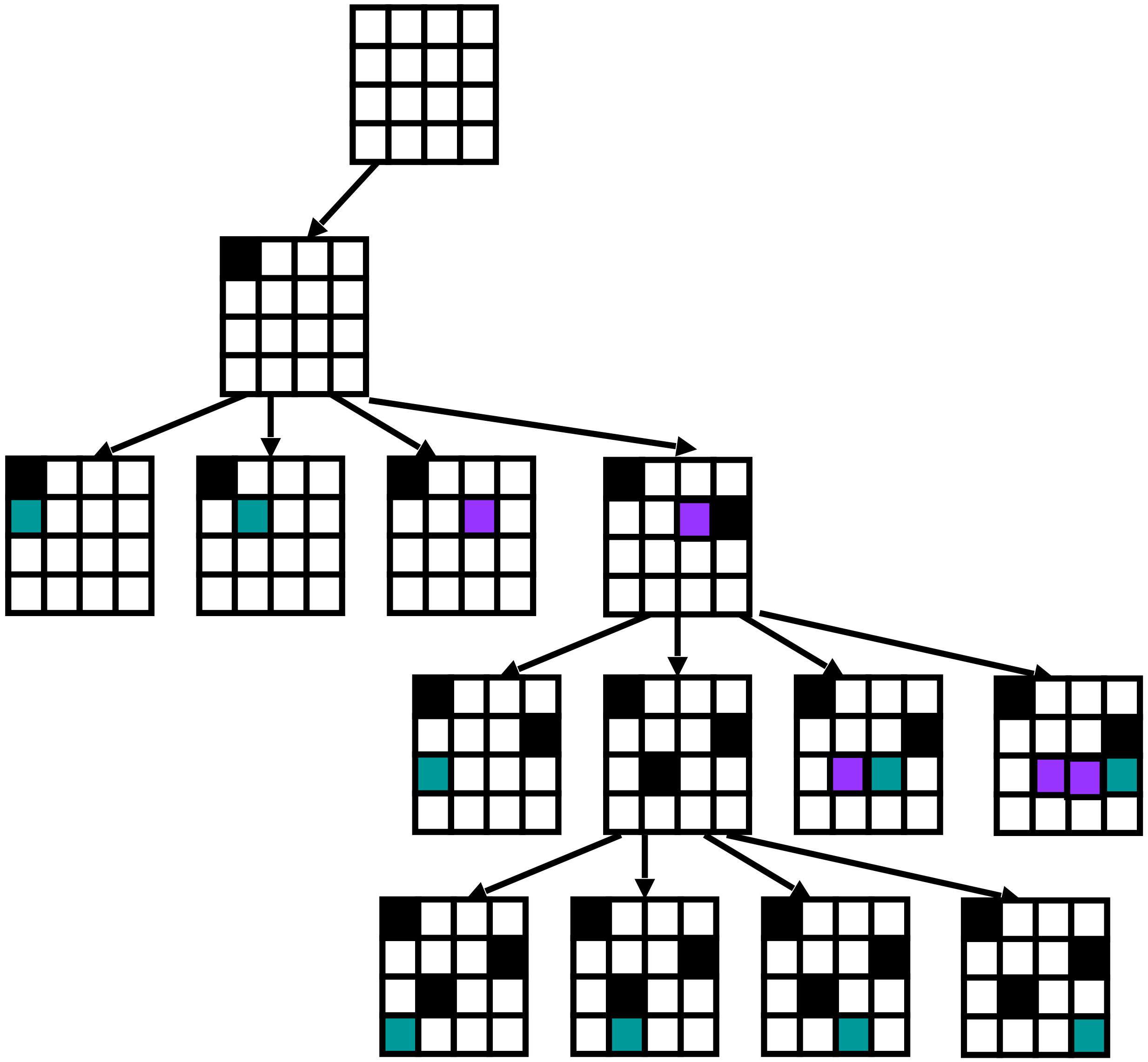
Example: N-Queens



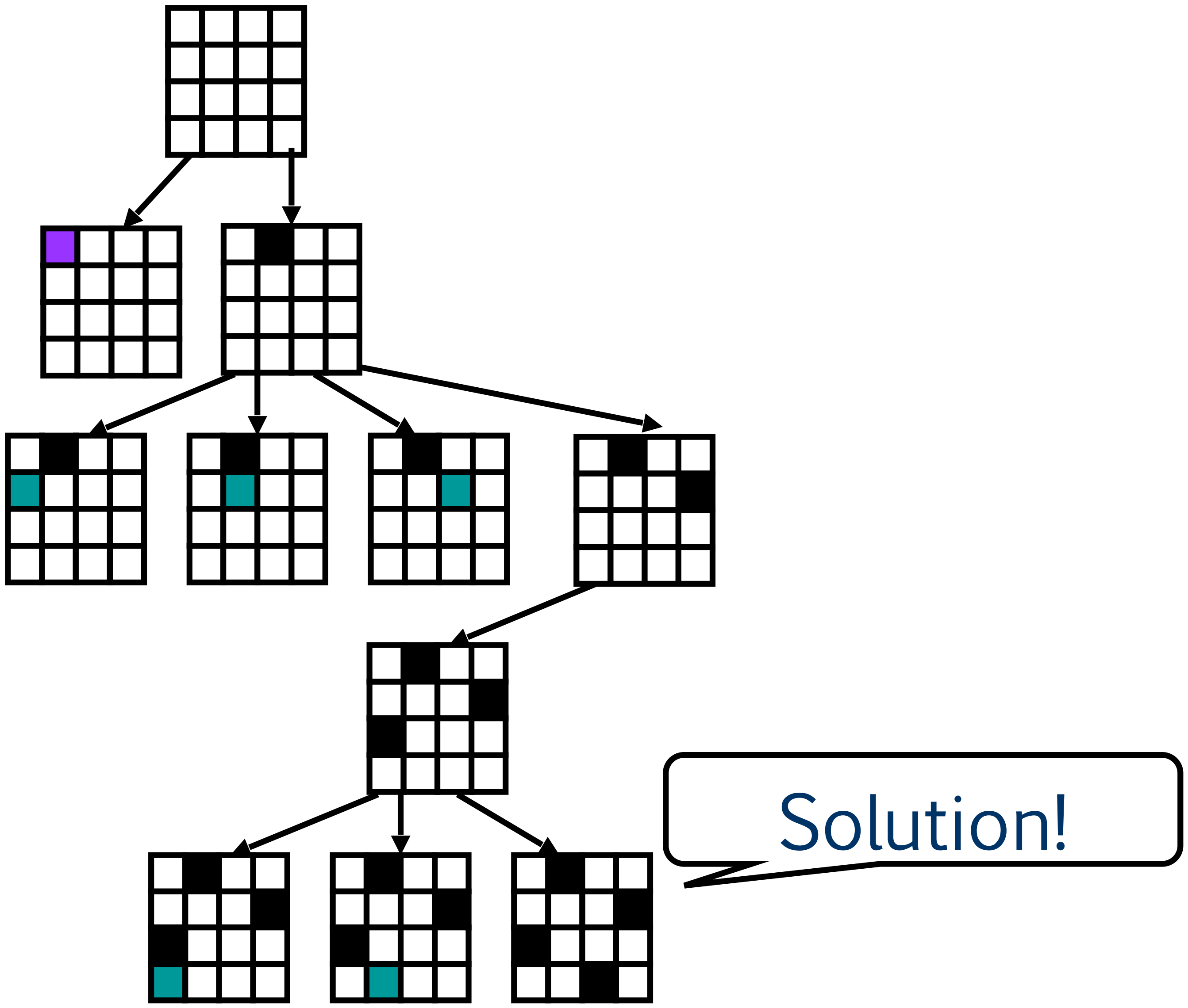
Example: N-Queens



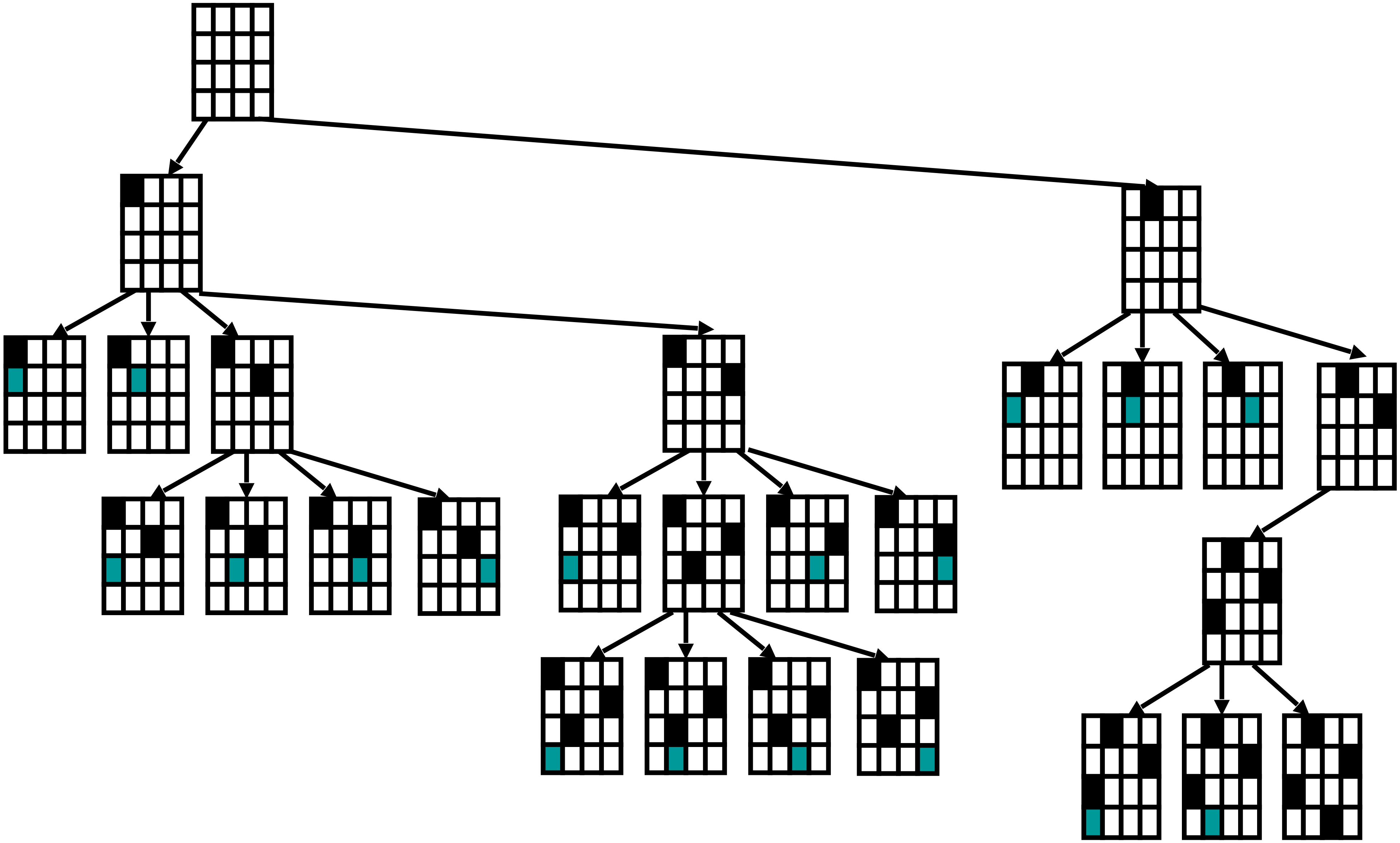
Example: N-Queens



Example: N-Queens



Example: N-Queens Backtracking Search Space



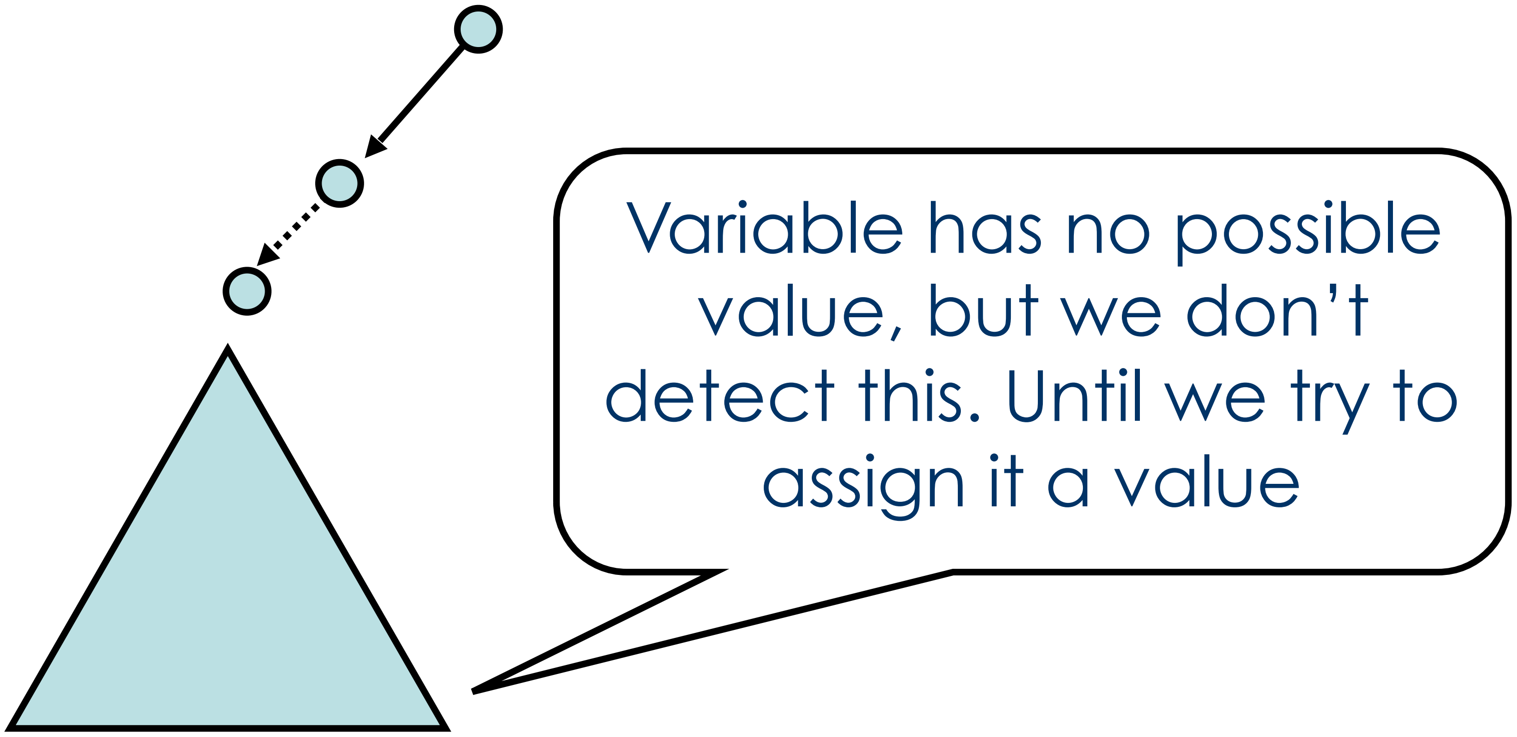
Problems with Plain Backtracking

Sudoku: The 3,3 cell has no possible value.

1	2	3						
		🟥				4	5	6
		7						
		8						
		9						

Problems with Plain Backtracking

- In the backtracking search we won't detect that the (3,3) cell has no possible value until all variables of the row/column (involving row or column 3) or the sub-square constraint (first sub-square) are assigned.
  So we have the following situation:


Variable has no possible value, but we don't detect this. Until we try to assign it a value

Constraint Propagation

- Constraint propagation refers to the technique of “looking ahead” at the yet unassigned variables in the search .
- Try to detect obvious failures: “Obvious” means 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 the search; potentially at every node of the search tree.
- Propagation itself is an inference step that needs some resources (in particular time)
  - If propagation is slow, this can slow the search down to the point where using propagation makes finding a solution take longer!
  - There is always a tradeoff between searching fewer nodes in the search, and having a higher nodes/ second processing rate.
- We will look at two main types of propagation: Forward Checking & Generalized Arc Consistency

Constraint Propagation: Forward Checking

- Forward checking is an extension of backtracking search that employs a "modest" amount of propagation (look ahead).
- 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 Algorithm

- For a single constraint C:

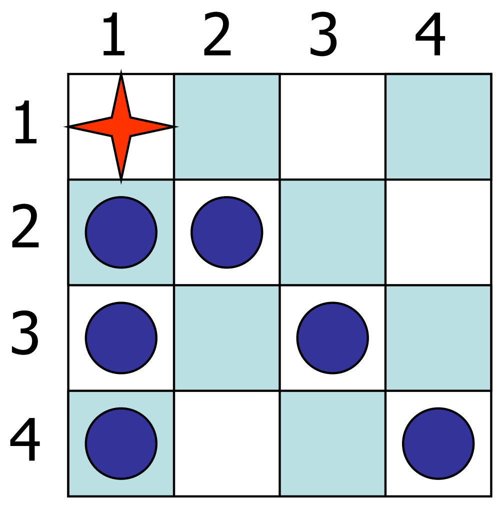
```
FCCheck(C,x)
  // C is a constraint with all its variables already
  // assigned, except for variable x.
  for d := each member of CurDom[x]
     IF making x = d together with previous assignments
        to variables in scope C falsifies C
     THEN remove d from CurDom[x]
 IF CurDom[x] = {} then return DWO (Domain Wipe Out)
 ELSE return ok
```

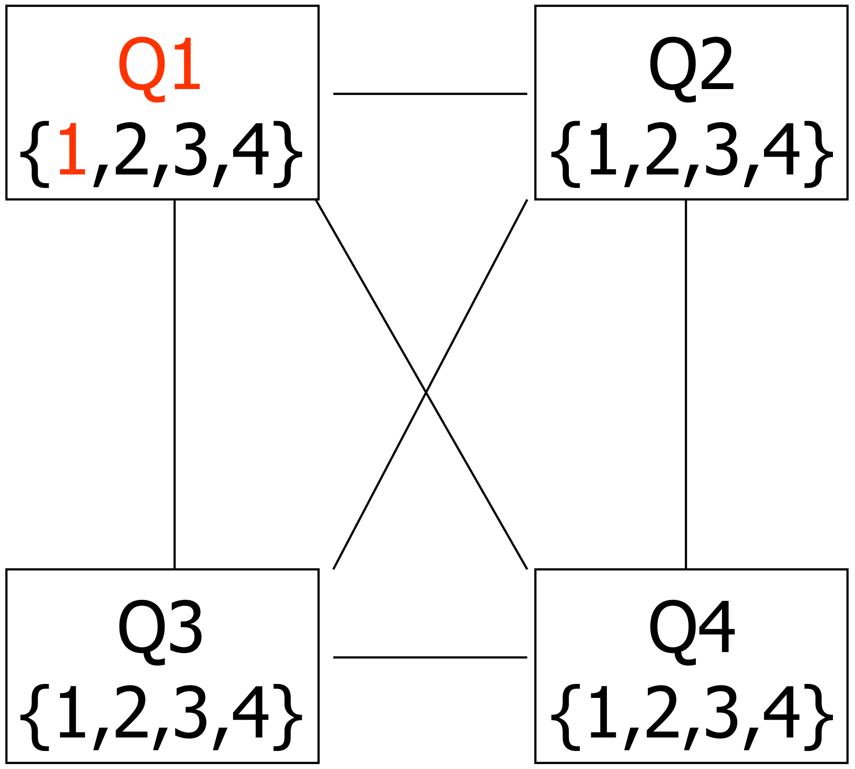
Forward Checking Algorithm

```
FC(Level) /*Forward Checking Algorithm */
   If all variables are assigned
        PRINT Value of each Variable
        RETURN or EXIT (RETURN for more solutions)
                       (EXIT for only one solution)
   V := PickAnUnassignedVariable()
   Assigned[V] := TRUE
   for d := each member of CurDom(V)
        Value[V] := d
        DWOoccured:= False
        for each constraint C over V such that
               a) C has only one unassigned variable X in its scope
           if(FCCheck(C,X) == DWO)    /* X domain becomes empty*/
                DWOoccurred:= True
                break /* stop checking constraints */
        if(not DWOoccured) /*all constraints were ok*/
           FC(Level+1)
        RestoreAllValuesPrunedByFCCheck()
   Assigned[V] := FALSE //undo since we have tried all of V's values
   return;
```

4-Queens Problem

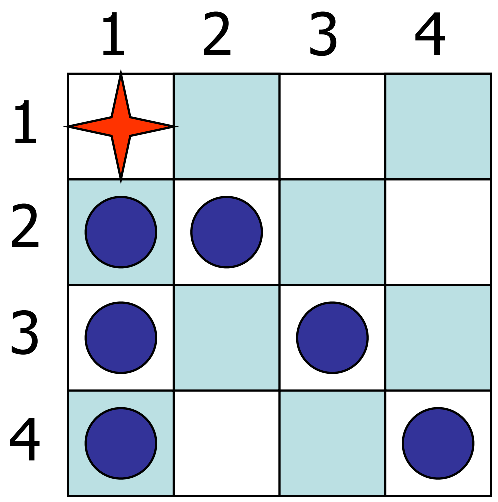
- Encoding with Q1, …, Q4 denoting a queen per row
  - cannot put two queens in same column

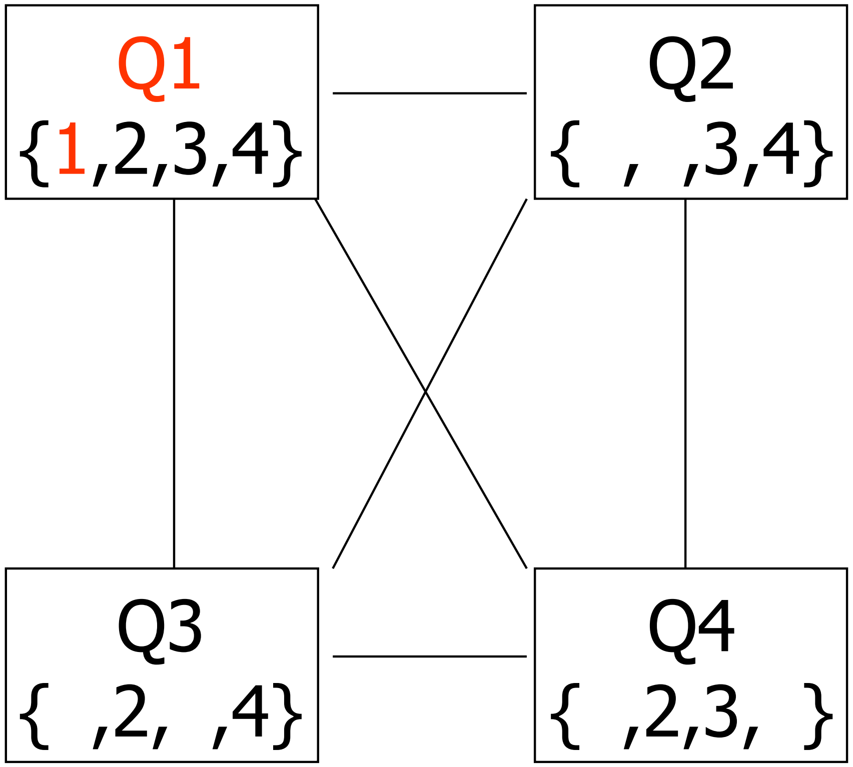




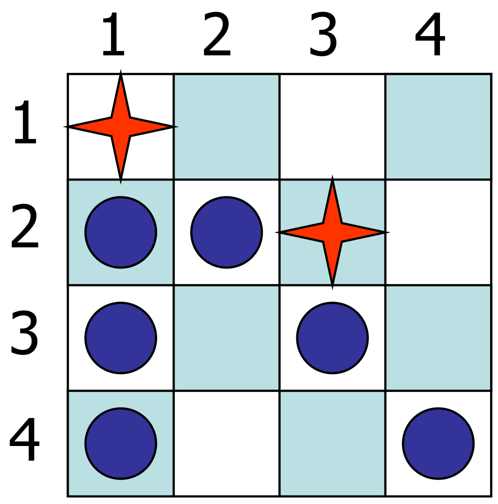
4-Queens Problem

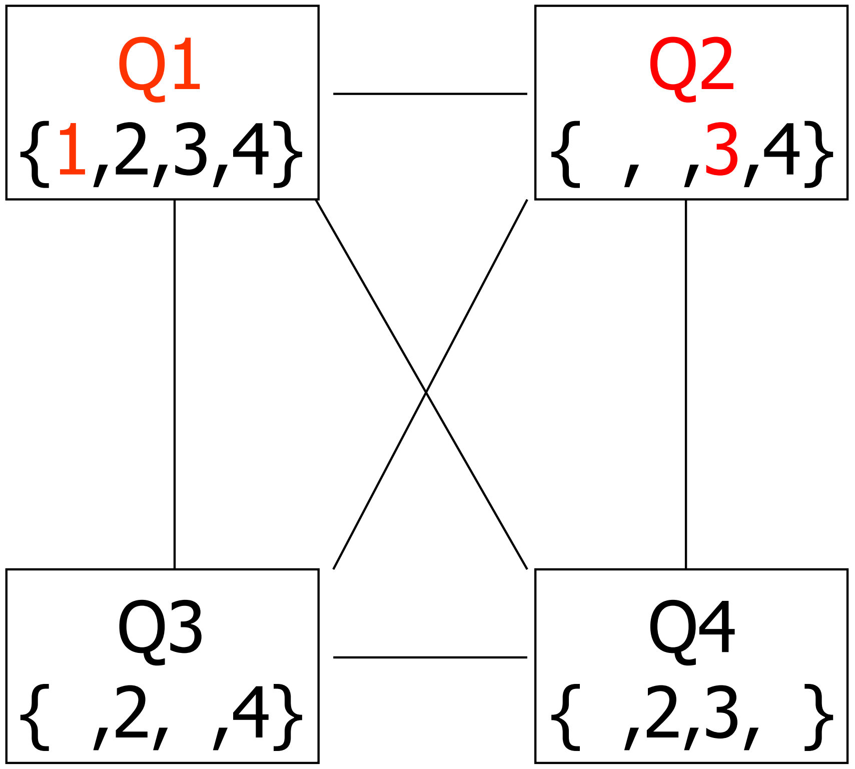
- Forward checking reduced the domains of all variables that are involved in a constraint with one uninstantiated variable:
  - Here all of Q2, Q3, Q4



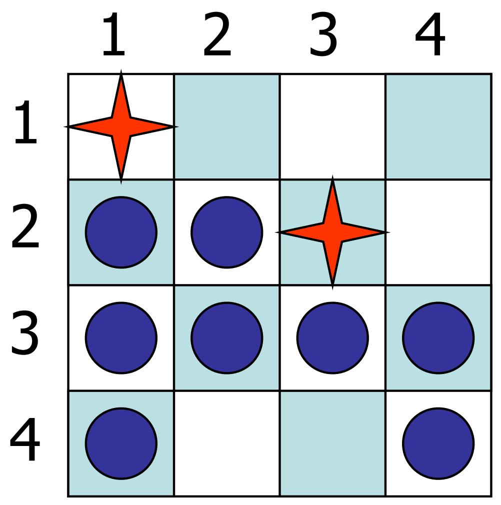


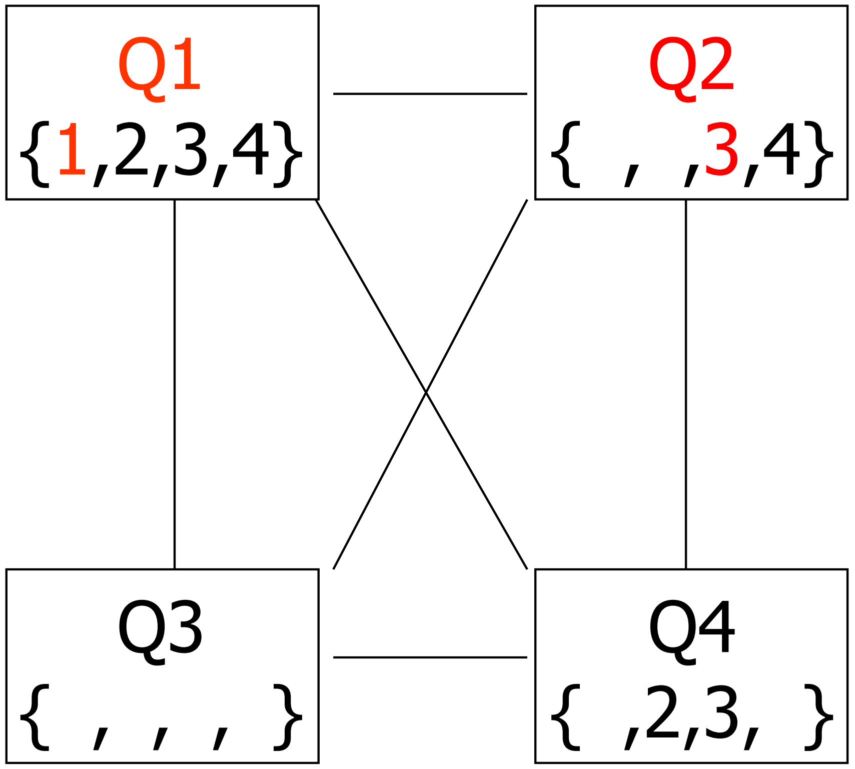
4-Queens Problem





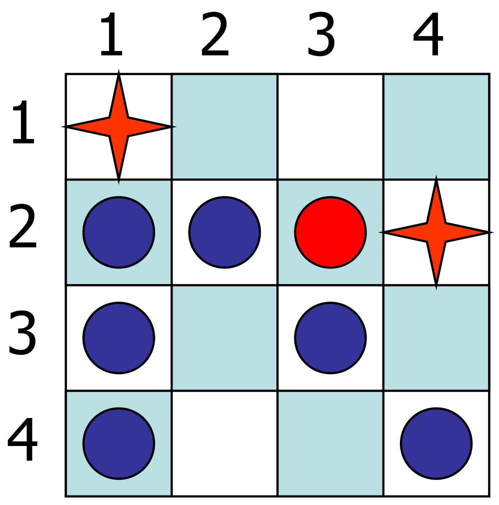
4-Queens Problem

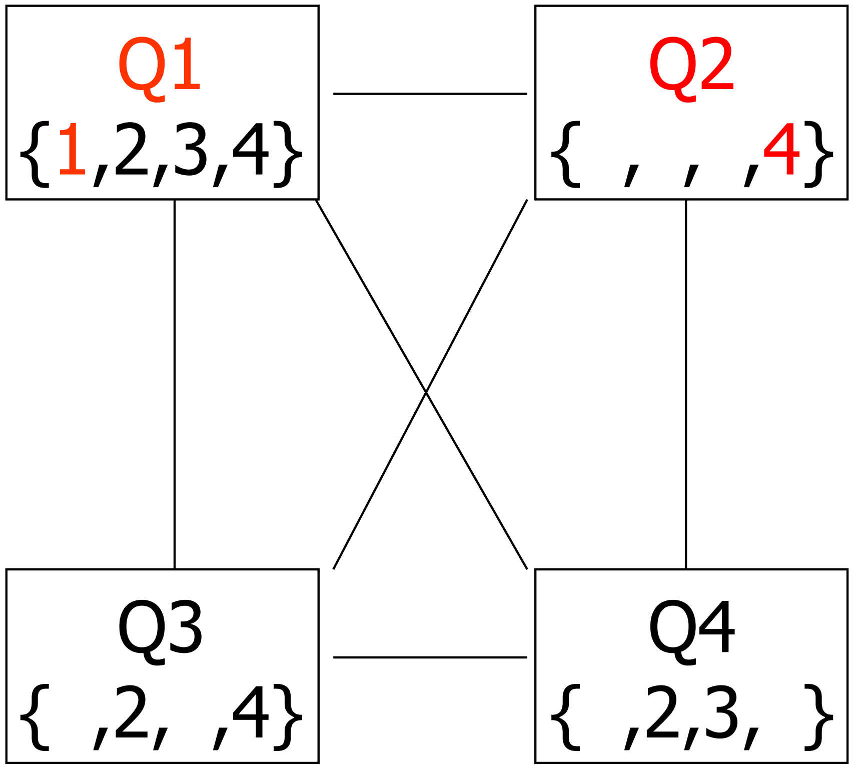




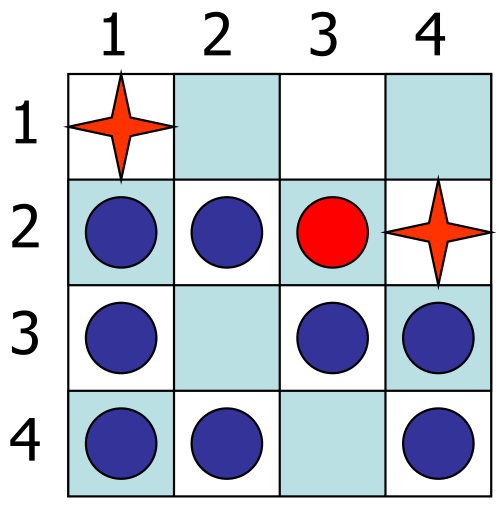
DWO

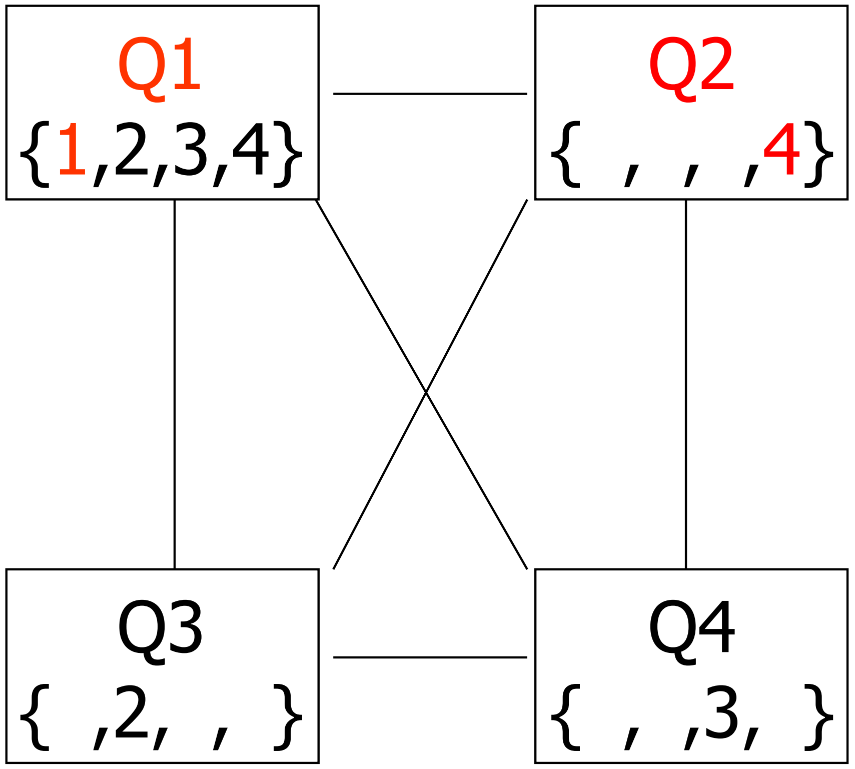
4-Queens Problem



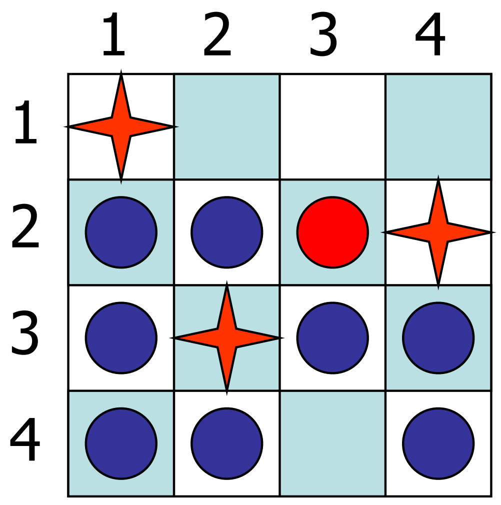


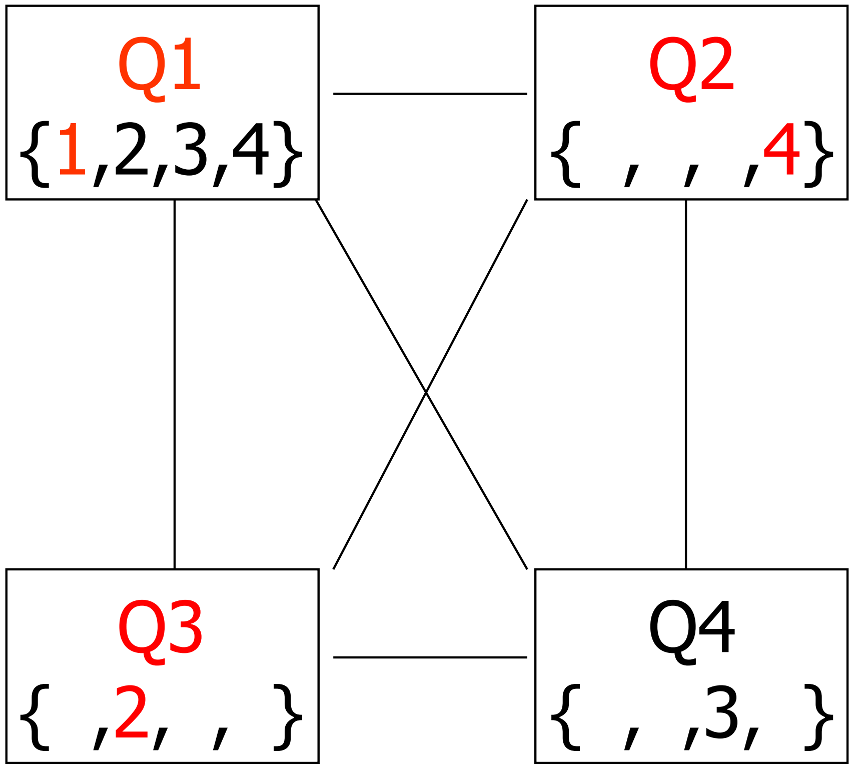
4-Queens Problem



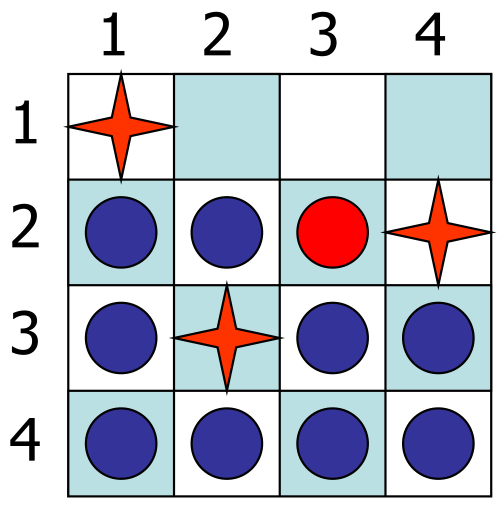


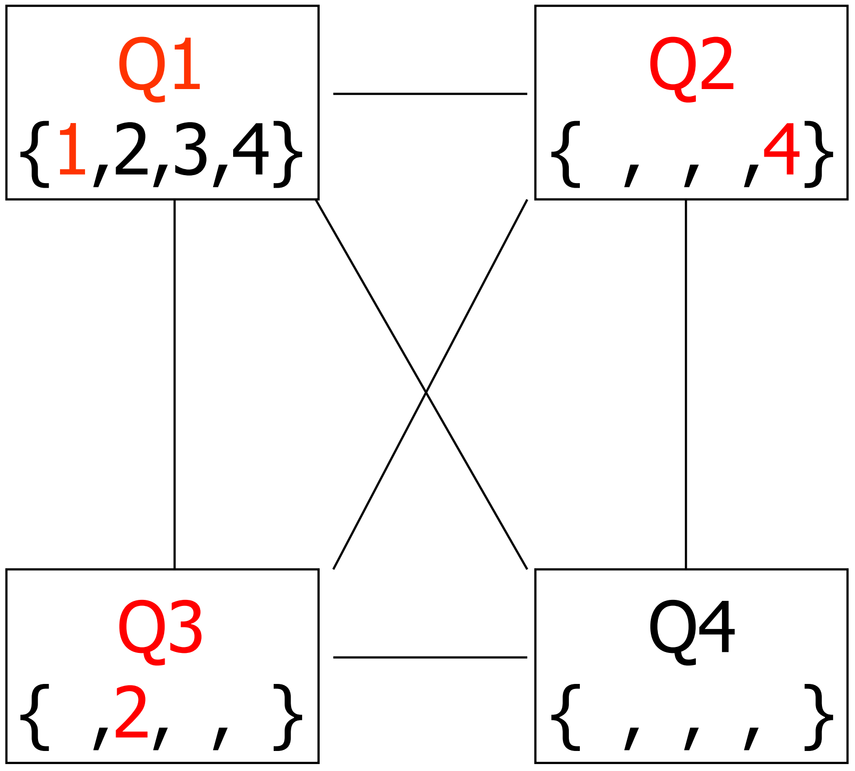
4-Queens Problem





4-Queens Problem

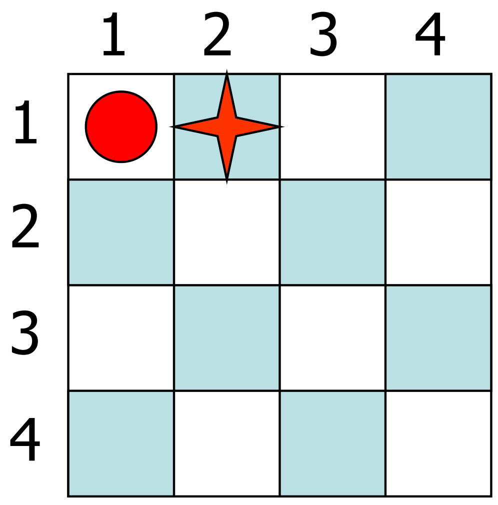


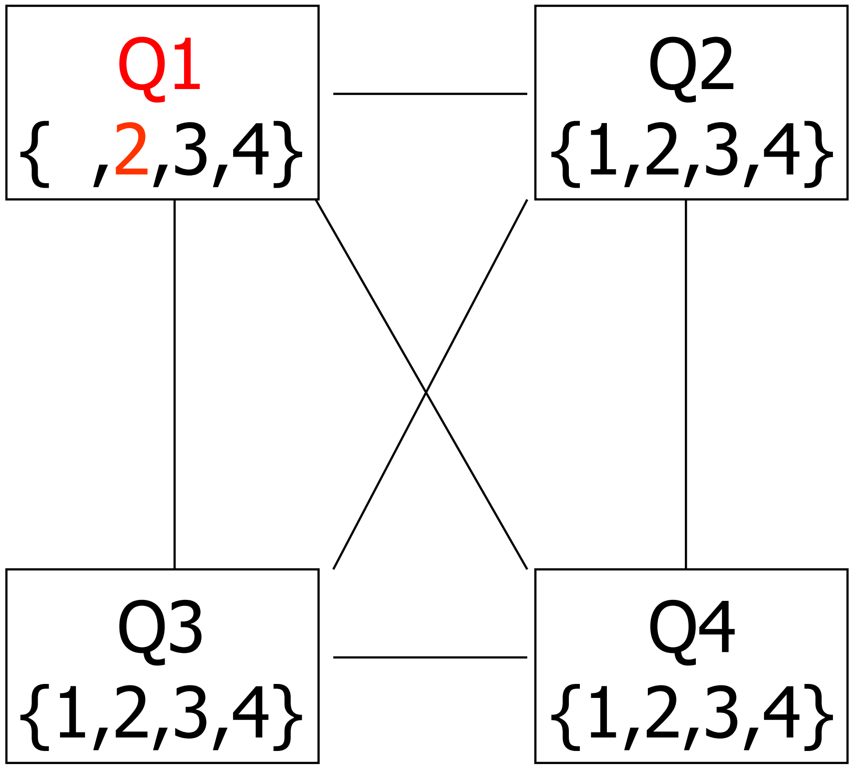


DWO

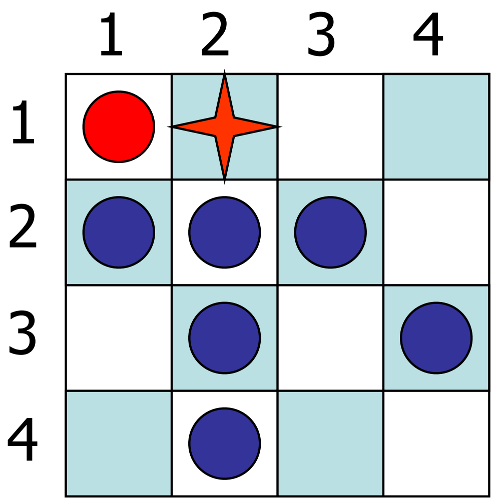
4-Queens Problem

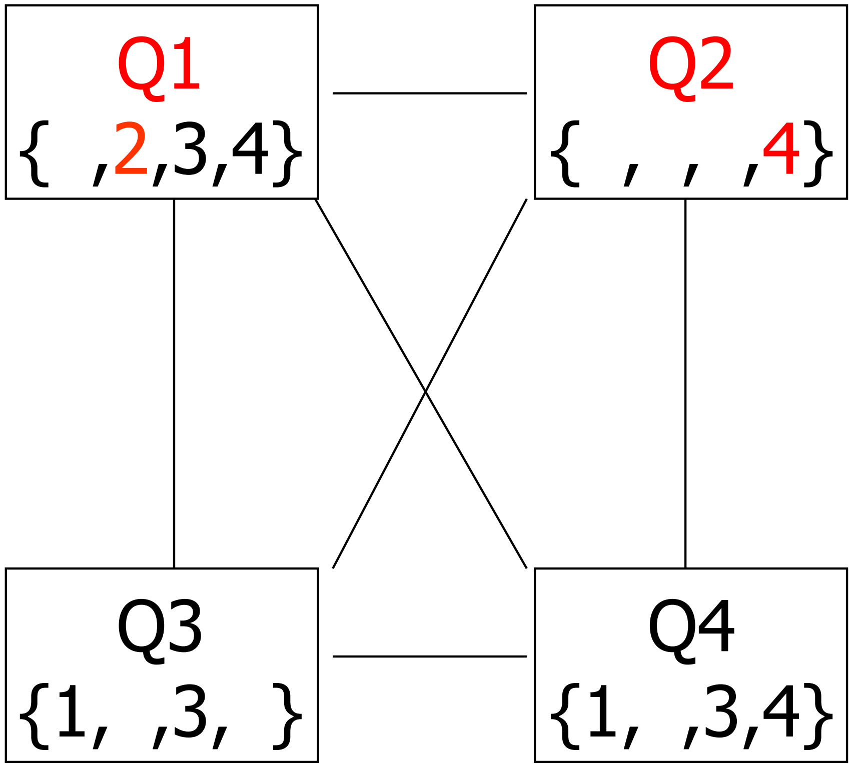
- Exhausted the subtree with Q1=1; try now Q1=2



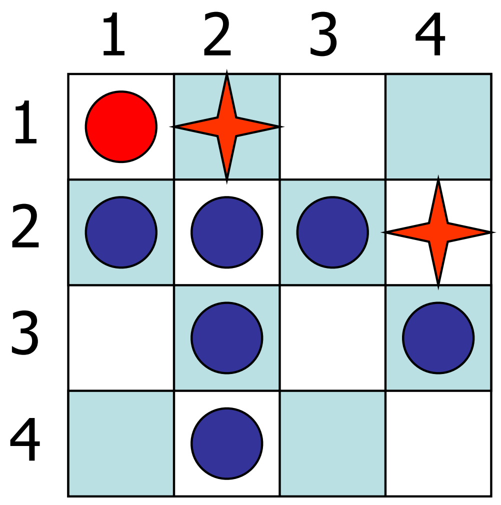


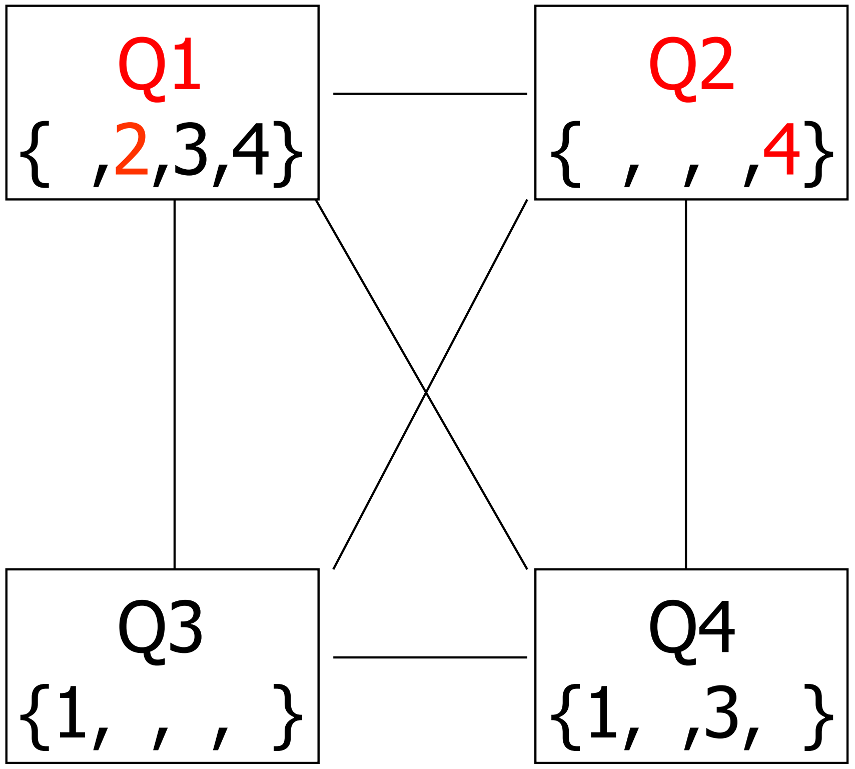
4-Queens Problem



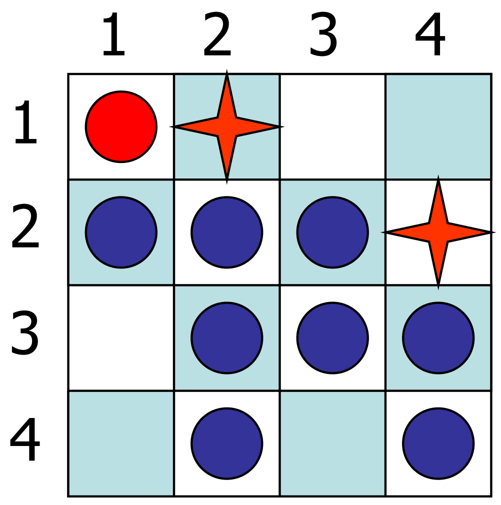


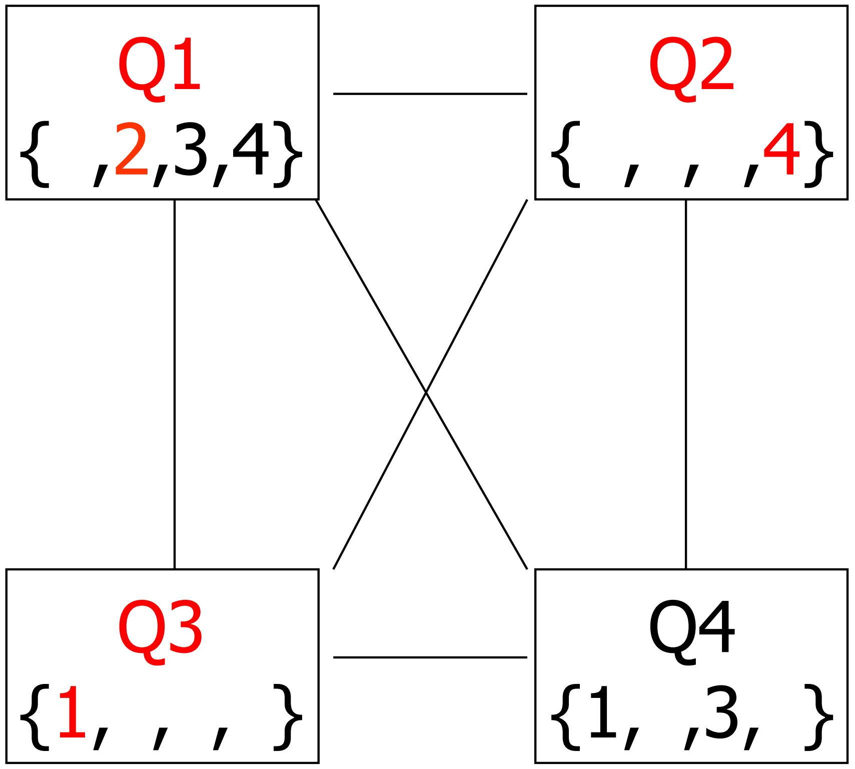
4-Queens Problem



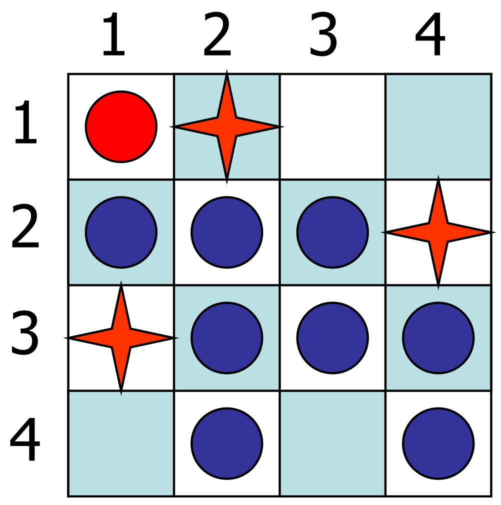


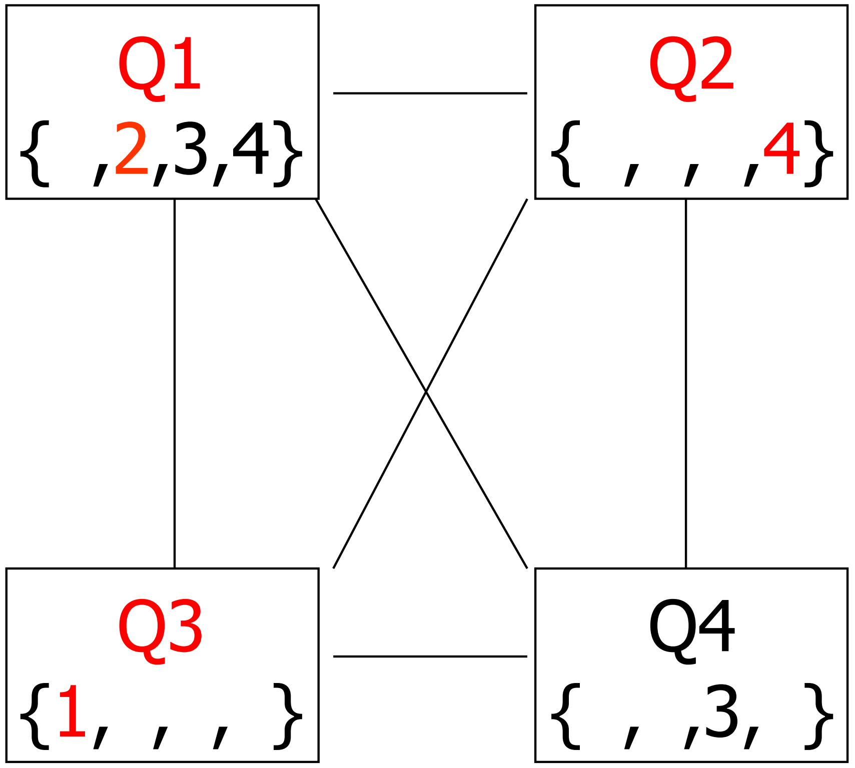
4-Queens Problem





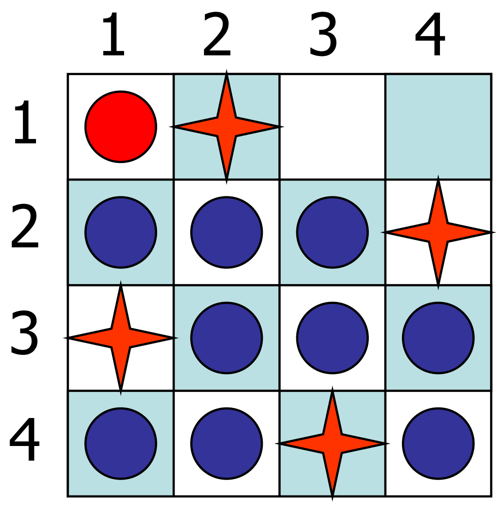
4-Queens Problem

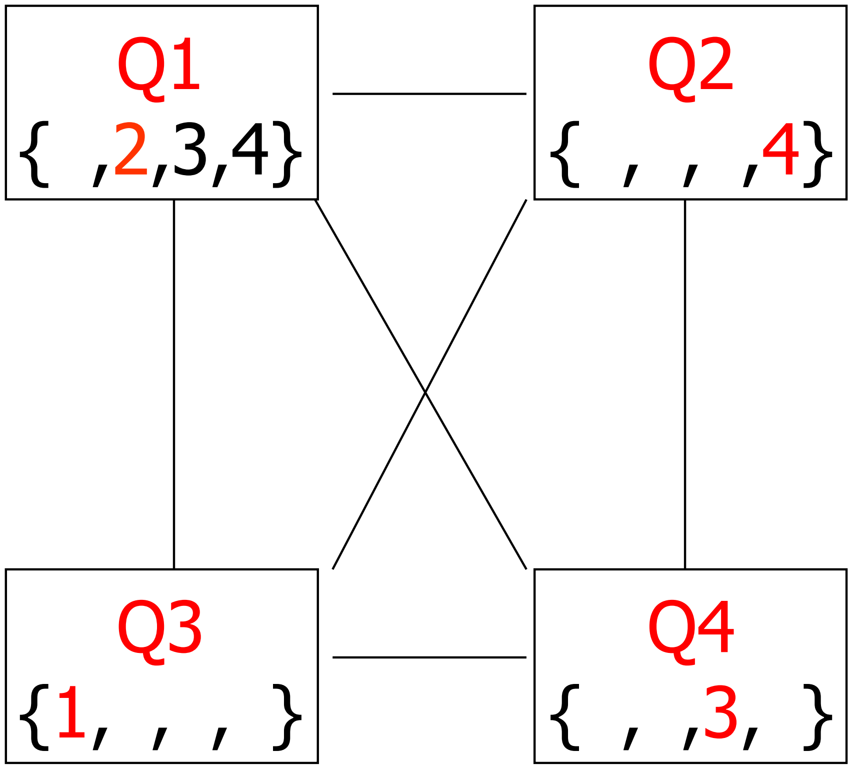




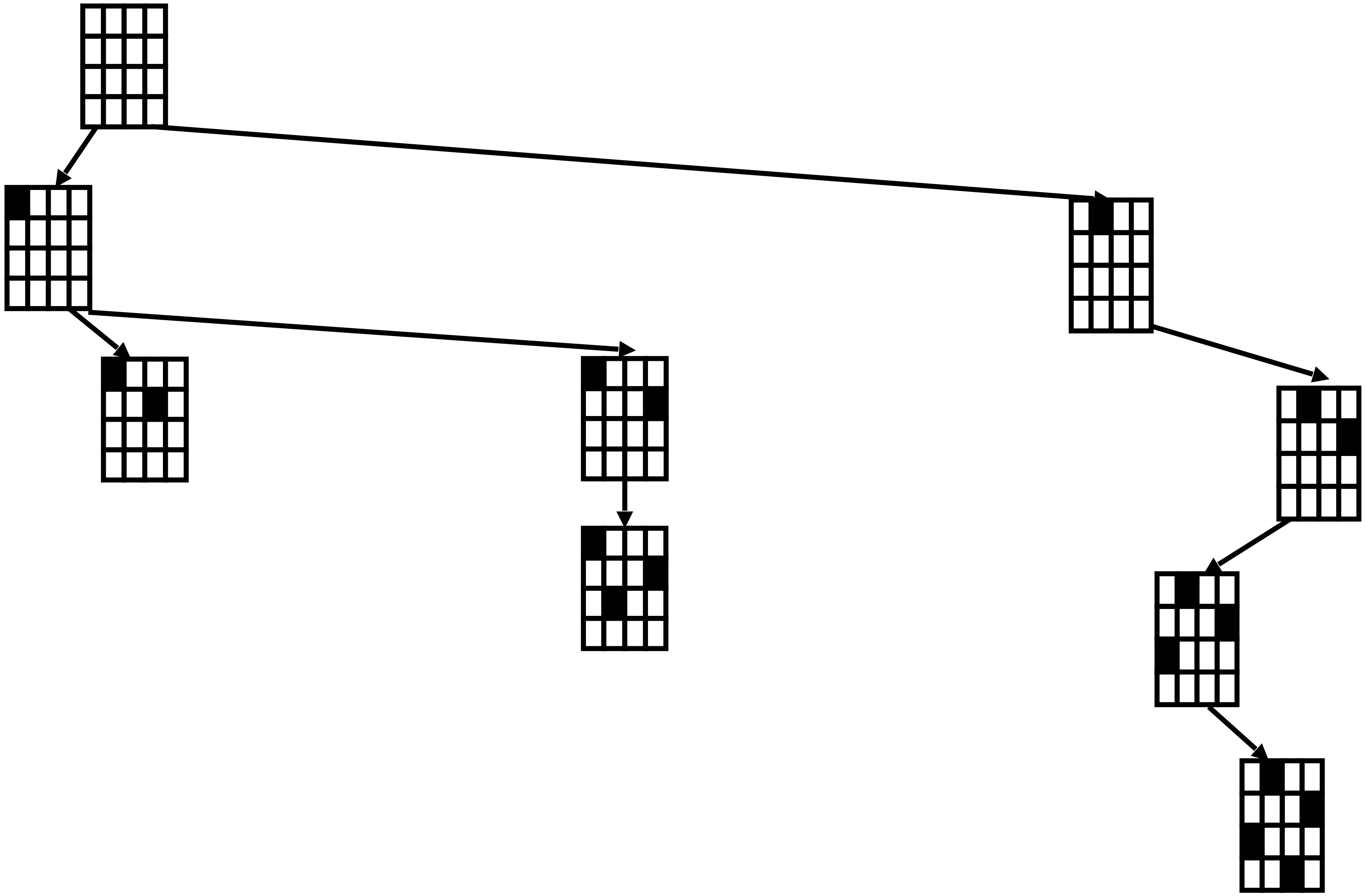
4-Queens Problem

- We have now find a solution: an assignment of all variables to values of their domain so that all constraints are satisfied





Example: N-Queens FC search Space



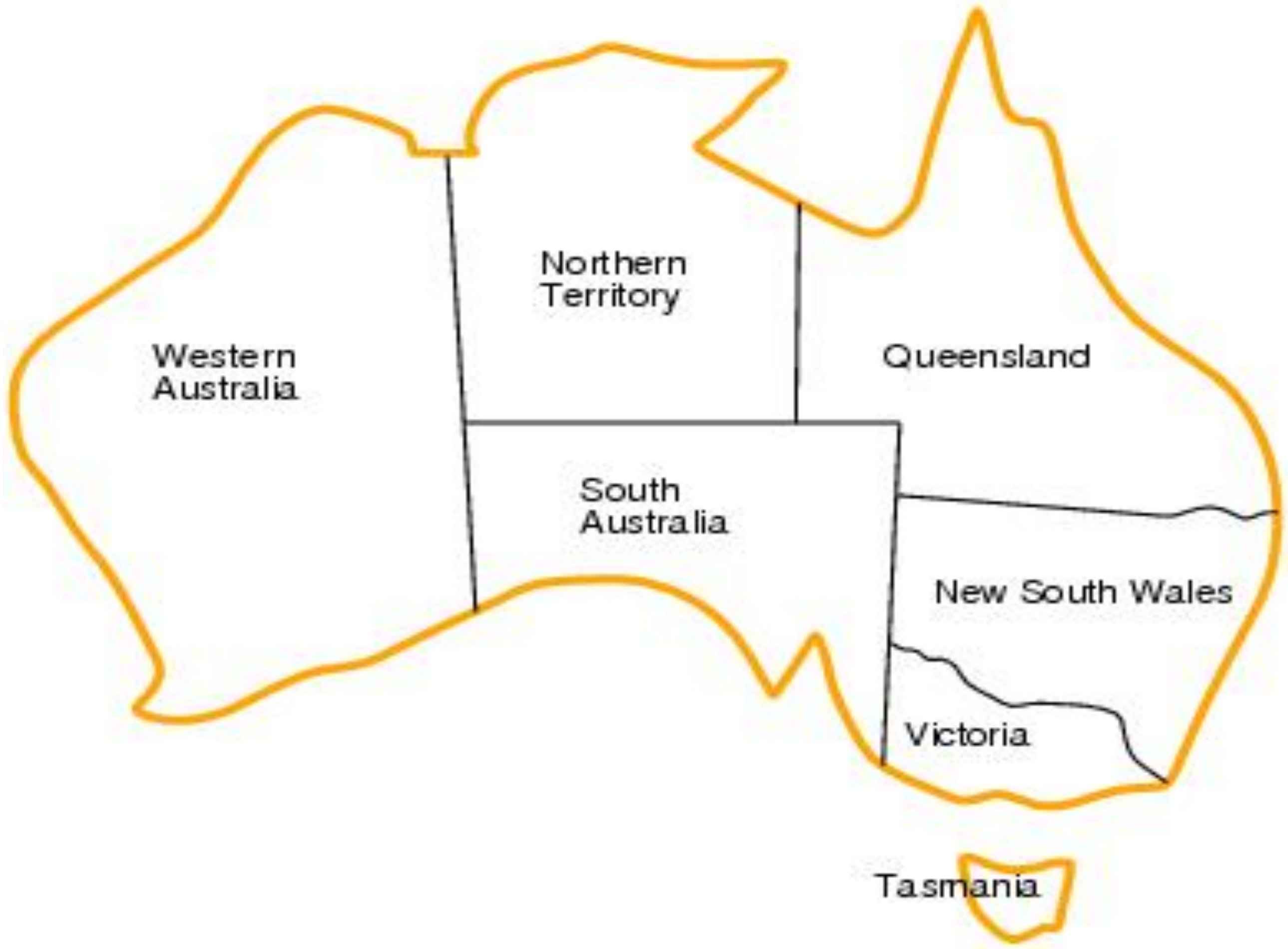
FC: Restoring Values

- After we backtrack from the 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 (FCCheck is called at every recursive invocation of FC).

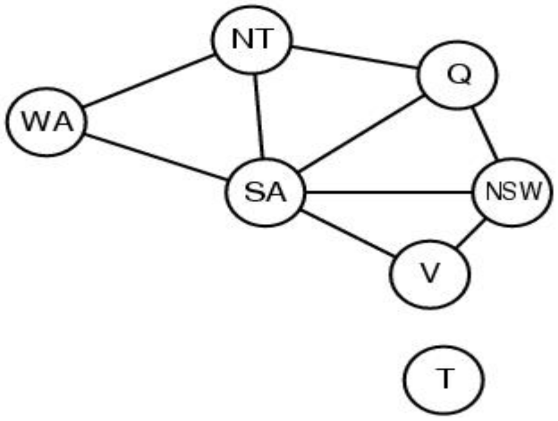
Example – Map Colouring

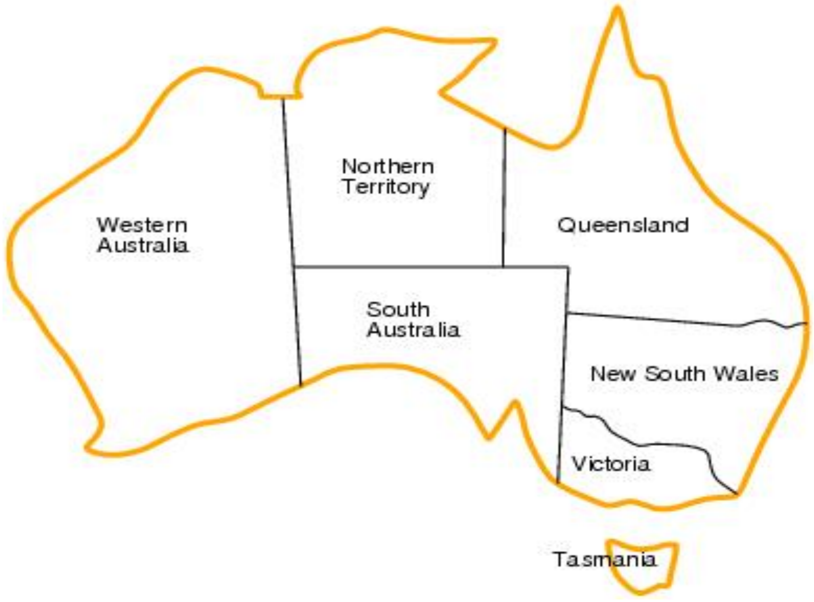
- Color the following map using *red*, *green*, and *blue* such that adjacent regions have different colours.



Example - Map Colouring

- **Modeling**
  - **Variables**: *WA, NT, Q, NSW, V, SA, T*
  - **Domains**: D_i=*{red, green, blue}*
  - **Constraints**: adjacent regions must have different colours.
    - E.g. $WA \neq NT$

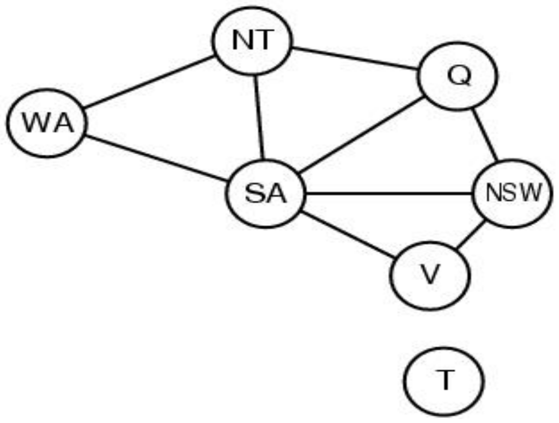


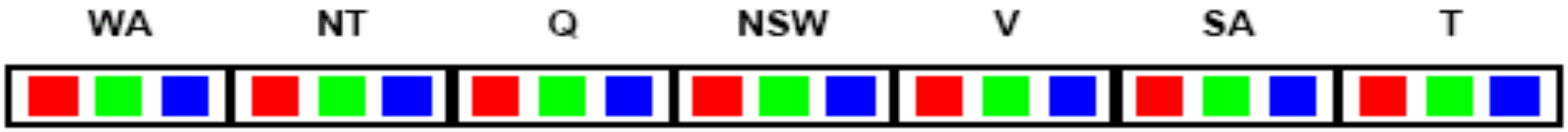


Example - Map Colouring

- *Forward checking idea*: keep track of remaining legal values for unassigned variables.

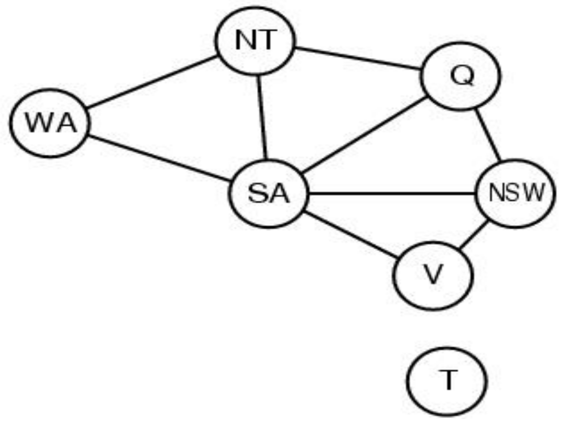
- Terminate search when any variable has no legal values.

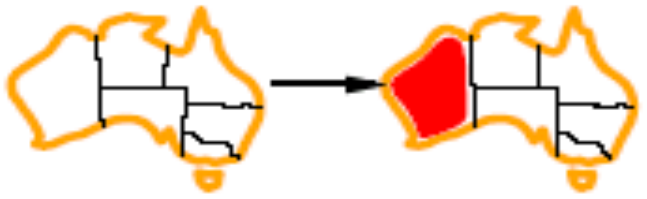


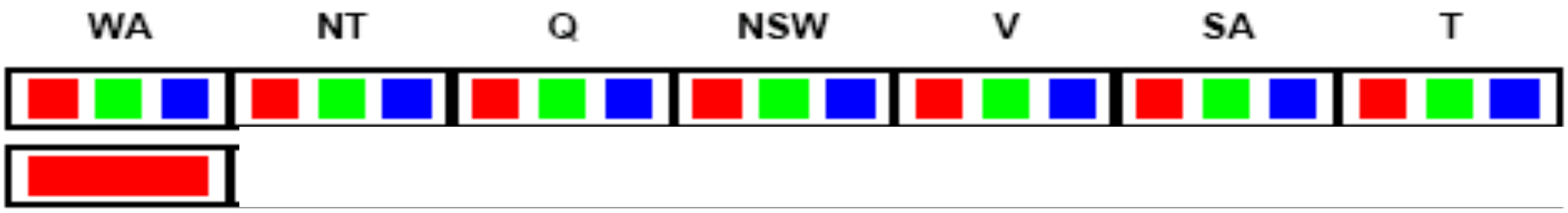


Example - Map Colouring

- Assign {*WA=red*}
- Effects on other variables connected by constraints to WA
  - *NT can no longer be red*
  - *SA can no longer be red*



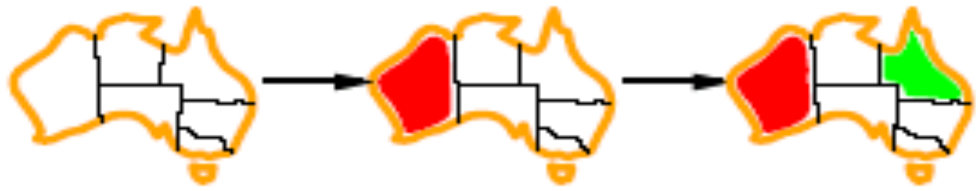


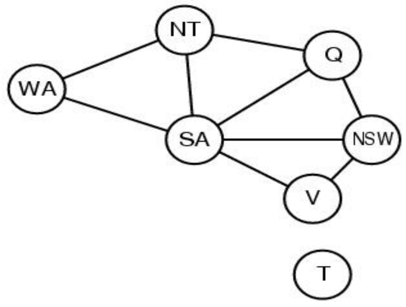


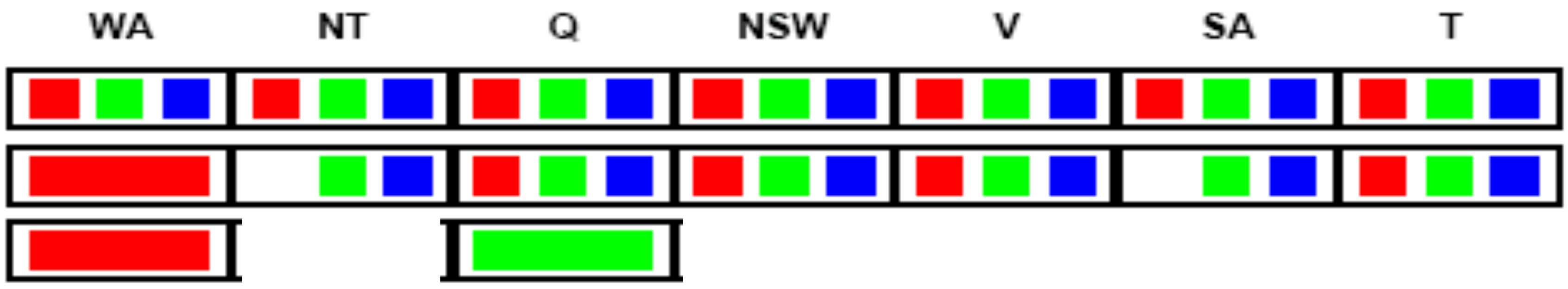
WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥						

Example – Map Colouring

- Assign *{Q=green} (Note: Not using MRV)*
- Effects on other variables connected by constraints with Q
  - *NT can no longer be green*
  - *NSW can no longer be green*
  - *SA can no longer be green*
- *MRV heuristic* would automatically select NT or SA next





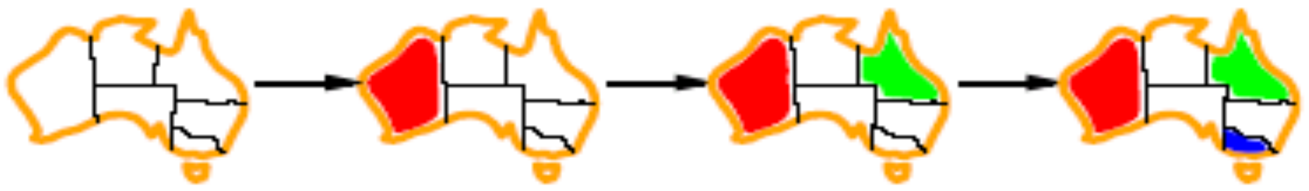


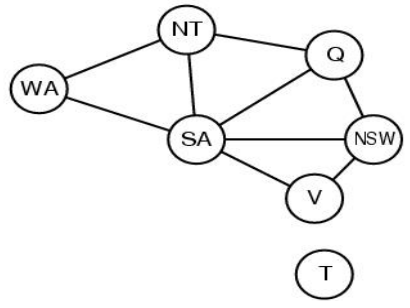
Example – Map Colouring

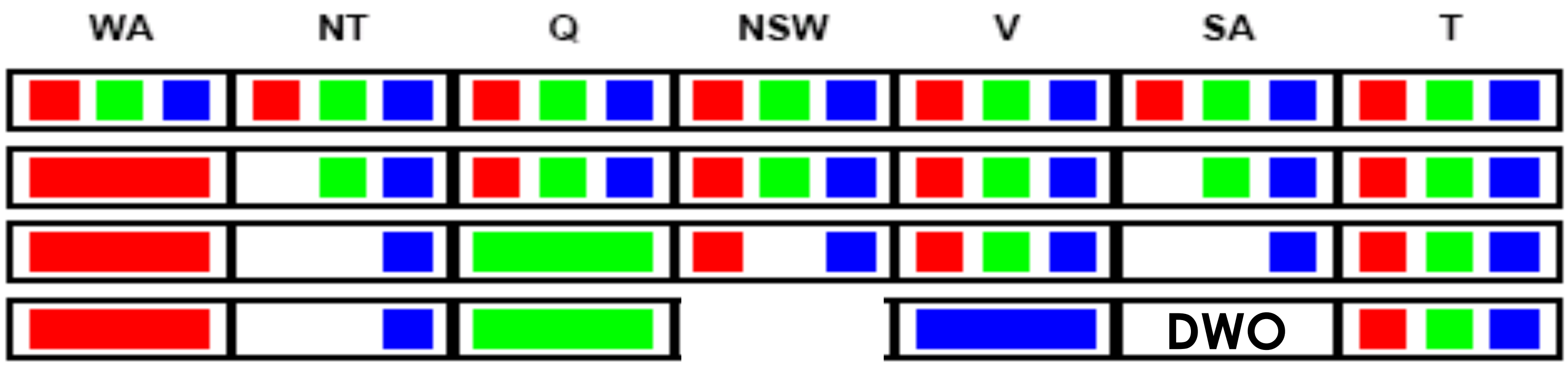
- Assign *{V=blue} (not using MRV)*

- Effects on other variables connected by constraints with V
  - *NSW can no longer be blue*
  - *SA is empty*

- FC has detected that partial assignment is *inconsistent* with the constraints and backtracking can occur.







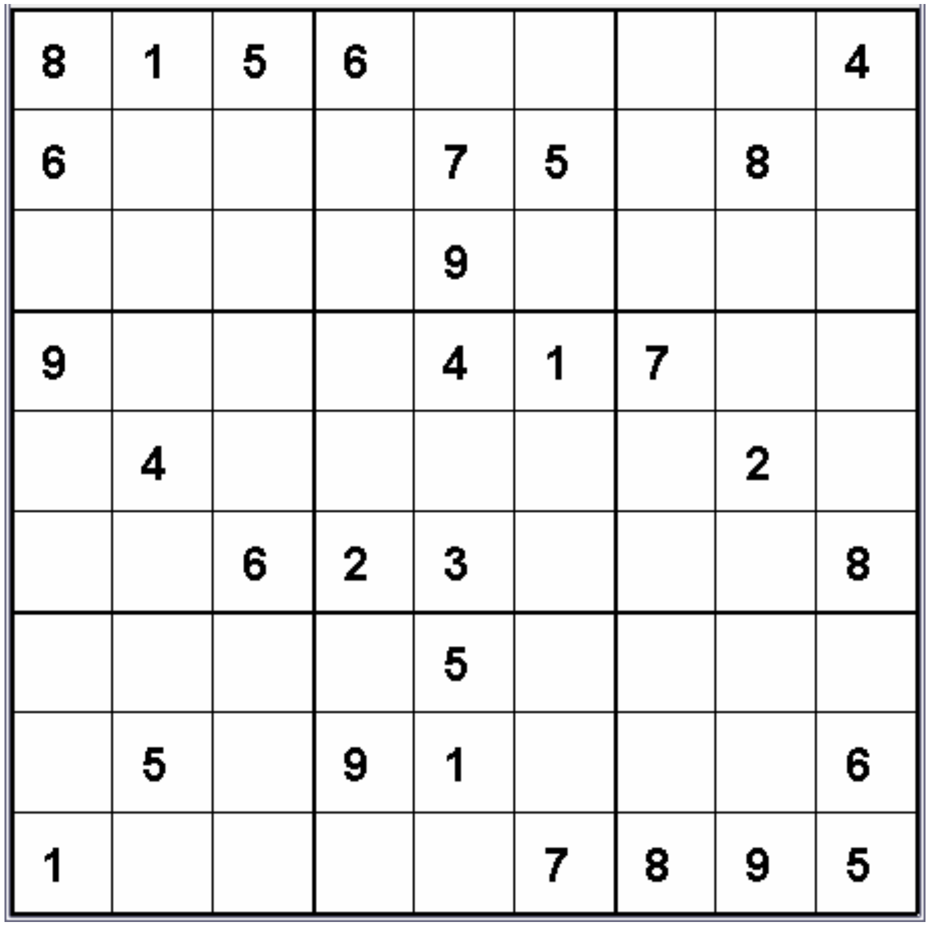
WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦
🟥	🟦	🟩		🟦	DWO	🟥🟩🟦

FC: Minimum Remaining Values Heuristics (MRV)

- FC also gives us for free a very powerful heuristic to guide us which variables to try next:
  - 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 when the tree starts to branch out (we start selecting variables with larger domains)
  - We can find a inconsistency much faster

MRV Heuristic: Human Analogy

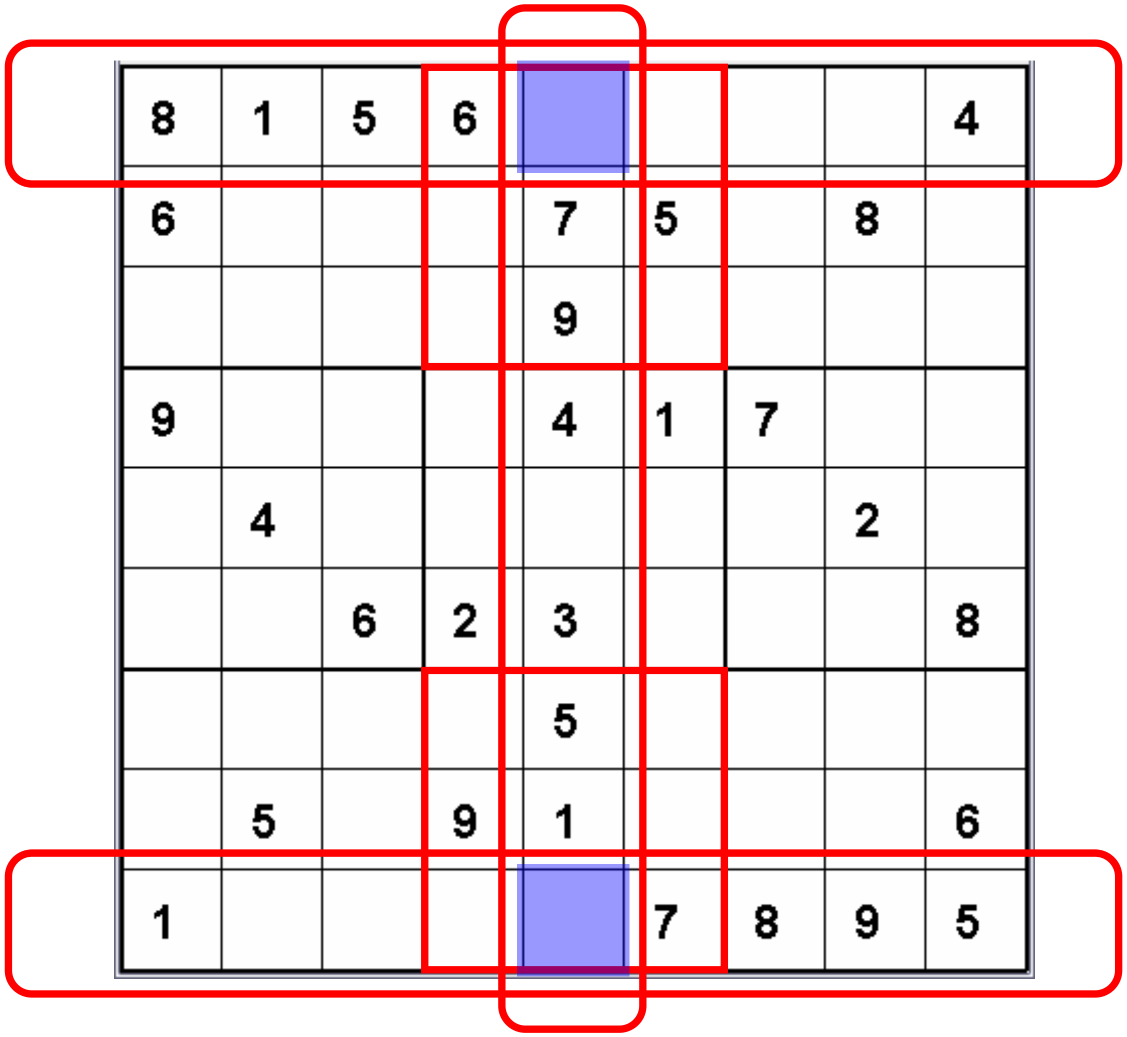
- What variables would you try first?



8	1	5	6					4
6				7	5		8	
				9				
9				4	1	7		
	4						2	
		6	2	3				8
				5				
	5		9	1				6
1					7	8	9	5

MRV Heuristic: Human Analogy

- What variables would you try first?



8	1	5	6	🟦				4
6				7	5		8	
				9				
9				4	1	7		
	4						2	
		6	2	3				8
				5				
	5		9	1				6
1				🟦	7	8	9	5

Domain of each variable: {1, …, 9}

(1, 5): impossible values:
Row: {1, 4, 5, 6, 8}
Column: {1, 3, 4, 5, 7, 9}
Subsquare: {5, 7, 9}
$\rightarrow$ Domain = {2}

(9, 5): impossible values:
Row: {1, 5, 7, 8, 9}
Column: {1, 3, 4, 5, 7, 9}
Subsquare: {1, 5, 7, 9}
$\rightarrow$Domain = {2, 6}

After assigning value 2 to cell (1,5): Domain = {6}

Most restricted variables! = MRV

Empirically

- FC often is about 100 times faster than BT
- FC with MRV (minimal remaining values) often 10000 times faster.
- But on some problems the speed up can be much greater
  - Converts problems that are not solvable to problems that are solvable.
- Still FC is not that powerful. Other more powerful forms of constraint propagation are used in practice.
- Try the previous map coloring example with MRV.

Constraint Propagation

- Another form of propagation is to make each arc in a constraint graph consistent
- C(X,Y) is **consistent** iff for every value of X there is some value of Y that satisfies C.
- We can use **inconsistencies** to remove values from the domains of variables:
    - e.g. C(X,Y): X>Y Dom(X)={1,5,11} Dom(Y)={3,8,15}
    - For X = 1 there is no value of Y s.t. Y is less than X. Remove 1 from the Domain of X
    - For Y = 15 there is no value of X s.t. X is more than Y. Remove 15 from the Domain of Y
    - We obtain Dom(X)={5,11} and D (Y)={3,8}
- Removing a value from a domain may trigger further inconsistency, so we have to repeat the procedure until everything is consistent.
- For efficient implementation, we keep track of inconsistency arcs by putting them in a Queue (See AC3 algorithm in the book)
- *This is stronger than forward checking. Why?*
- Checking for consistency can be done as a pre-processing step, or it can be directly integrated into a search algorithm.

Constraint Propagation: Generalized Arc Consistency

GAC—Generalized Arc Consistency

1. $C(V_1, V_2, V_3, \ldots, V_n)$ is GAC with respect to variable Vi, if and only if

   For **every** value of V_i, there exists values of $V_1, V_2, V_{i-1}, V_{i+1}, \ldots, V_n$ that satisfy C.

   Note that the values are removed from the variable domains during search. So variables that are GAC in a constraint might become inconsistent (non-GAC).

Constraint Propagation: Generalized Arc Consistency

- $C(V_1, V_2, V_3, \ldots, V_n)$ is GAC if and only if

  It is GAC with respect to every variable in its scope.

- A CSP is GAC if and only if

  all of its constraints are GAC.

Constraint Propagation: Arc Consistency

- Say we find a value **d** of variable **V_i** that is not consistent: That is, there is no assignments to the other variables that satisfy the constraint when **V_i = d**
  - d is said to be Arc Inconsistent
  - We **can remove d** from the domain of Vi—this value cannot lead to a solution (much like Forward Checking, but more powerful).

- Remember the example of C(X,Y): X>Y Dom(X)={1,5,11} Dom(Y)={3,8,15}
  - **Arc inconsistent values were pruned from domains.**

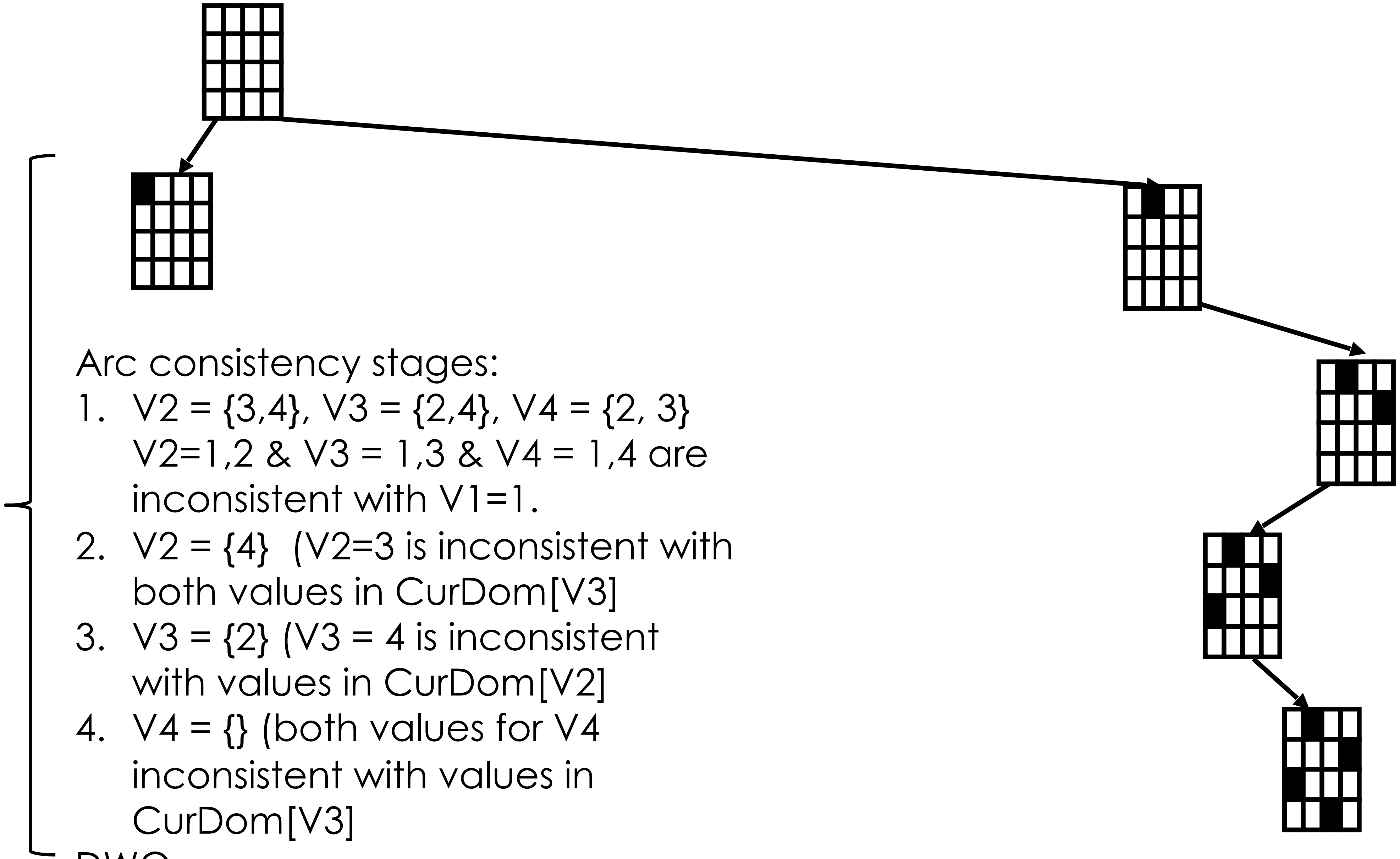
Constraint Propagation: Arc Consistency

- If we apply arc consistency propagation during search the search tree's size will typically be much reduced in size.

- Removing a value from a variable domain may trigger further inconsistency, so we have to repeat the procedure until everything is consistent.

    – We put constraints on a queue and add new constraints to the queue as we need to check for arc consistency.

Example: N-Queens GAC search Space

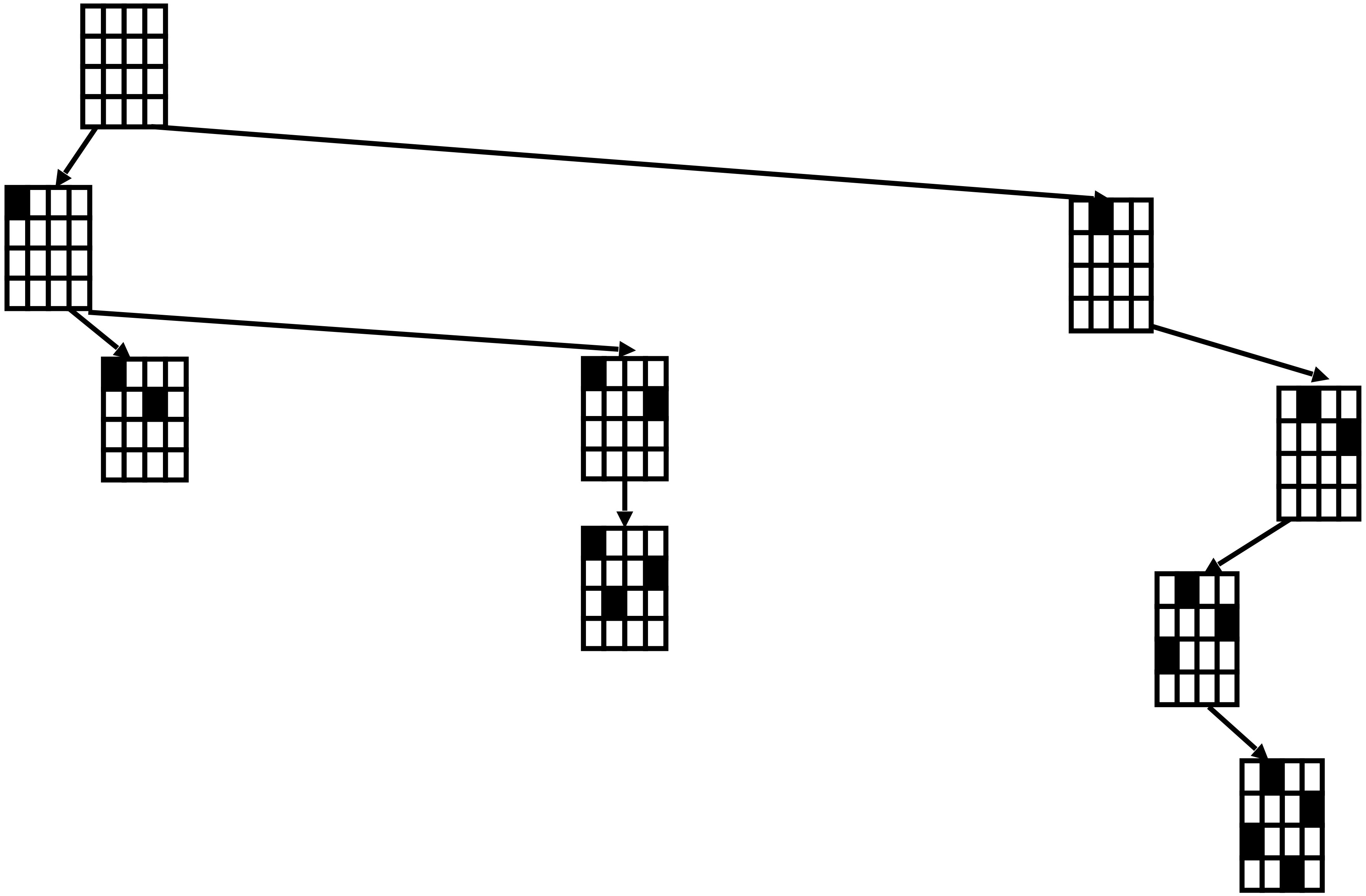


Arc consistency stages:

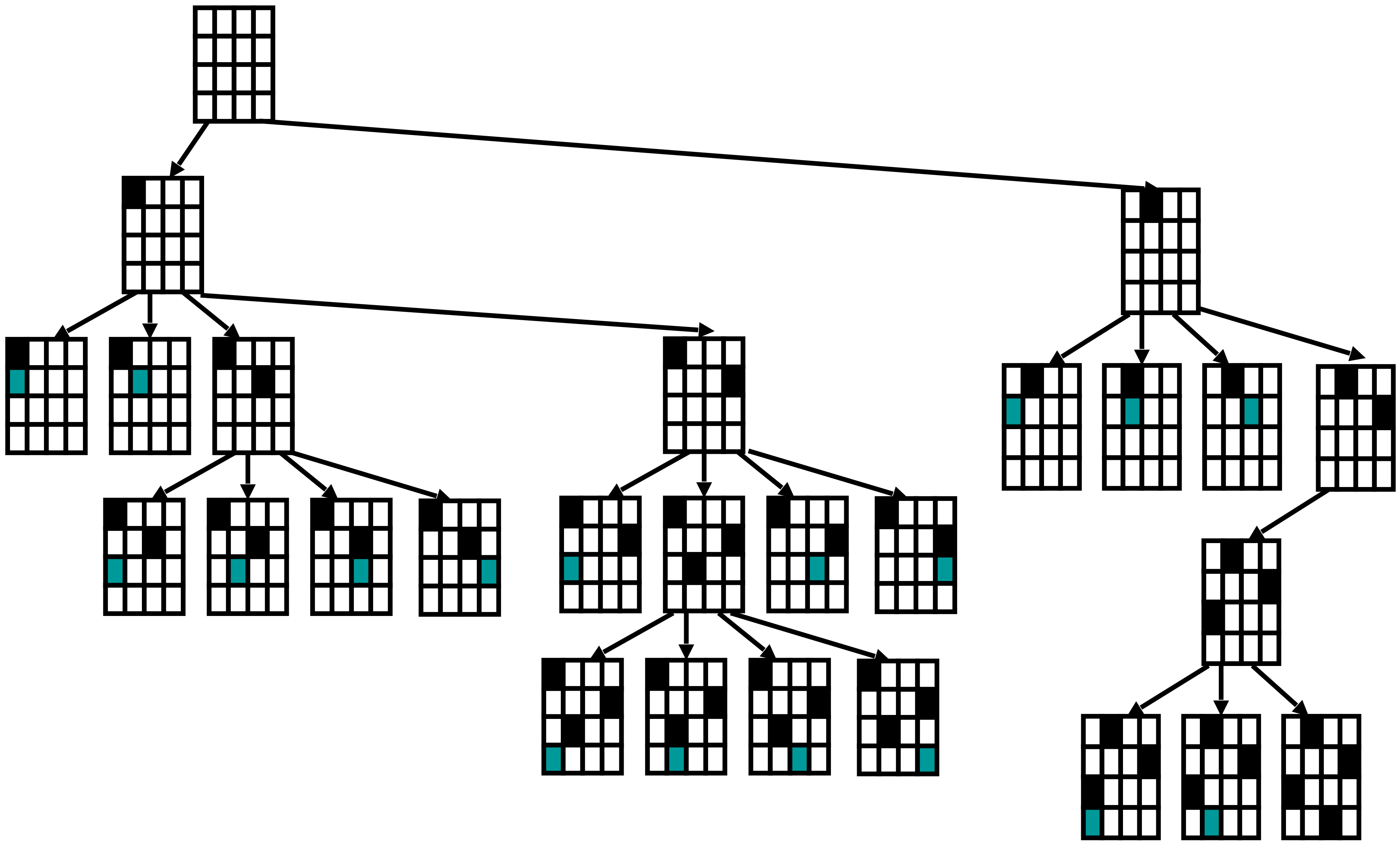
1. V2 = {3,4}, V3 = {2,4}, V4 = {2, 3}
   V2=1,2 & V3 = 1,3 & V4 = 1,4 are inconsistent with V1=1.
2. V2 = {4} (V2=3 is inconsistent with both values in CurDom[V3]
3. V3 = {2} (V3 = 4 is inconsistent with values in CurDom[V2]
4. V4 = {} (both values for V4 inconsistent with values in CurDom[V3]

DWO

Example: N-Queens FC search Space

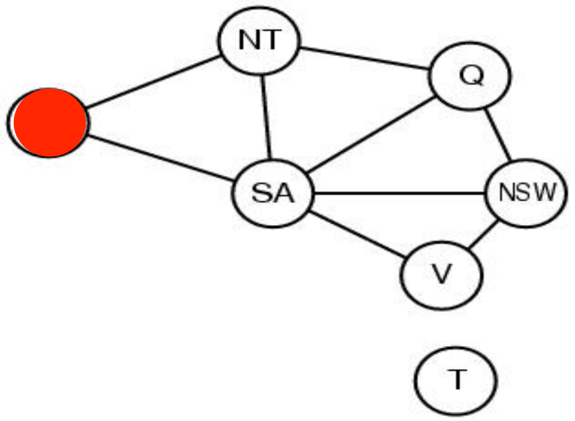


Example: N-Queens Backtracking Search Space



Example - Map Colouring

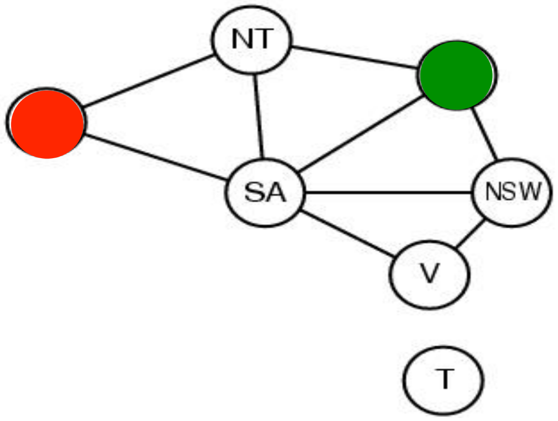
- Assign *{WA=red}*
- Effects on other variables connected by constraints to WA
  - *NT can no longer be red = {G, B}*
  - *SA can no longer be red = {G, B}*
- *All other values are arc-consistent*



Example - Map Colouring

- Assign {Q=green}
- Effects on other variables connected by constraints with Q
  - *NT can no longer be green = {B}*
  - *NSW can no longer be green = {R, B}*
  - *SA can no longer be green = {B}*
- *DWO there is no value for SA that will be consistent with NT ≠ SA and NT = B*

Note Forward Checking would not have detected this DWO.



GAC Algorithm

- We make all constraints GAC at every node of the search space.
- This is accomplished by removing from the domains of the variables all arc inconsistent values.

GAC Algorithm, enforce GAC during search

```
GAC(Level) /*Maintain GAC Algorithm */
   If all variables are assigned
      PRINT Value of each Variable
      RETURN or EXIT (RETURN for more solutions)
                     (EXIT for only one solution)
   V := PickAnUnassignedVariable()
   Assigned[V] := TRUE
   for d := each member of CurDom(V)
      Value[V] := d
      Prune all values of V ≠ d from CurDom[V]
      for each constraint C whose scope contains V
        Put C on GACQueue
      if(GAC_Enforce() != DWO)
        GAC(Level+1) /*all constraints were ok*/
      RestoreAllValuesPrunedFromCurDoms()
   Assigned[V] := FALSE
   return;
```

Enforce GAC (prune all GAC inconsistent values)

```
GAC_Enforce()
   // GAC-Queue contains all constraints one of whose variables has
   // had its domain reduced. At the root of the search tree
   // first we run GAC_Enforce with all constraints on GAC-Queue

   while GACQueue not empty
       C = GACQueue.extract()
       for V := each member of scope(C)
           for d := CurDom[V]
                Find an assignment A for all other
                variables in scope(C) such that
                C(A ∪ V=d) = True
                if A not found
                    CurDom[V] = CurDom[V] - d
                    if CurDom[V] = ∅
                          empty GACQueue
                          return DWO //return immediately
                    else
                          push all constraints C' such that
                          V ∈ scope(C') and C' ∉ GACQueue
                          on to GACQueue
   return TRUE //while loop exited without DWO
```

Enforce GAC

- A support for V=d in constraint C is an assignment **A** to all of the other variables in scope(C) such that A U {V=d} satisfies C. (**A** is what the algorithm's inner loop looks for).

- Smarter implementations keep track of "supports" to avoid having to search though all possible assignments to the other variables for a satisfying assignment.

Enforce GAC

- Rather than search for a satisfying assignment to C containing V=d, we check to see if the current support is still valid: i.e., all values it assigns still lie in the variable's current domains

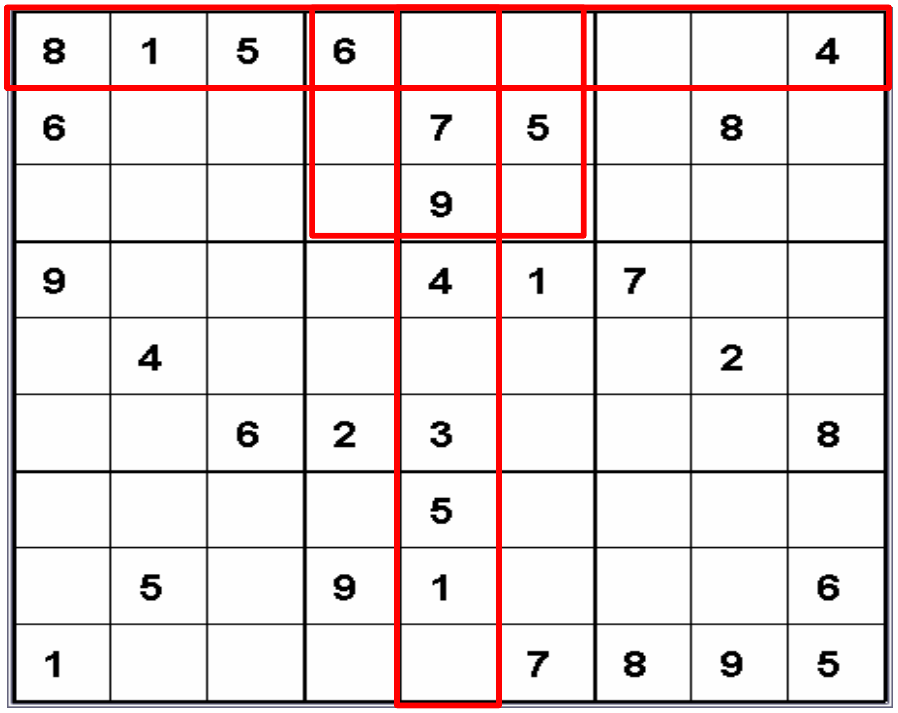
- Also we take advantage that a support for V=d, e.g.
  {V=d, X=a, Y=b, Z=c}
  is also a support for X=a, Y=b, and Z=c

Enforce GAC

- However, finding a support for V=d in constraint C still in the worst case requires O(2^k) work, where k is the arity of C, i.e., |scope(C)|.

- Another key development in practice is that for some constraints this computation can be done in polynomial time.
  E.g., all-diff(V1, …. Vn) we can be check if Vi=d has a support in the current domains of the other variables in polynomial time using ideas from graph theory.
    We do not need to examine all combinations of values for the other variables looking for a support

GAC enforce example



□ = All-diff

C_{SS2} = All-diff($V_{1,4}$, $V_{1,5}$, $V_{1,6}$, $V_{2,4}$, $V_{2,5}$, $V_{2,6}$, $V_{3,4}$, $V_{3,5}$, $V_{3,6}$)

C_{R1} = All-diff($V_{1,1}$, $V_{1,2}$, $V_{1,3}$, $V_{1,4}$, $V_{1,5}$, $V_{1,6}$, $V_{1,7}$, $V_{1,8}$, $V_{1,9}$)

C_{C5} = All-diff($V_{1,5}$, $V_{2,5}$, V_{35}, $V_{4,5}$, $V_{5,5}$ $V_{6,5}$, $V_{7,5}$, $V_{8,5}$, $V_{9,5}$)

By going back and forth between constraints we get more values pruned.

GAC(C_{SS2}) ➔ CurDom of $V_{1,5}$, $V_{1,6}$, $V_{2,4}$, $V_{3,4}$, $V_{3,6}$ = {1, 2, 3, 4, 8}

GAC(C_{R1}) ➔ CurDom of $V_{1,7}$, $V_{1,8}$ = {2, 3, 7, 9}

CurDom of $V_{1,5}$, $V_{1,6}$, = {2, 3}

GAC(C_{SS2}) ➔ CurDom of $V_{2,4}$, $V_{3,4}$, $V_{3,6}$ = {1, 4, 8}

GAC(C_{C5}) ➔ CurDom of $V_{5,5}$,$V_{9,5}$ = {2, 6, 8}

➔ CurDom of $V_{1,5}$ = {2}

GAC(C_{SS2}) ➔ CurDom of $V_{1,6}$ = {3}

Many real-world applications of CSP

- Assignment problems
  - who teaches what class
- Timetabling problems
  - exam schedule
- Transportation scheduling
- Floor planning
- Factory scheduling
- Hardware configuration
  - a set of compatible components