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.

Representing States with Feature Vectors

- For each problem we have designed a new state representation (and coded the sub-routines called by search to use 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

- We have
  - A set of k variables (or features)
  - 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.

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 (but not all) cells filled in

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 a legal setting of the variables 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

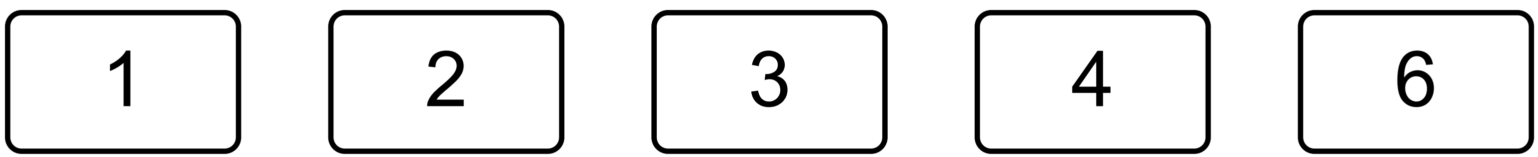
Constraint Satisfaction Problems

- 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 setting of the variables that satisfies the goal.
  - A setting of the variables that satisfies some constraints.
- In contrast, in the 8-puzzle, the setting of the variables satisfying the goal is given. We care about the sequence of moves needed to move the tiles into that configuration.
  - In Search we care about finding a path to the goal state.
  - In CSPs we care about finding the goal state (not how to get there).

Example Car Sequencing

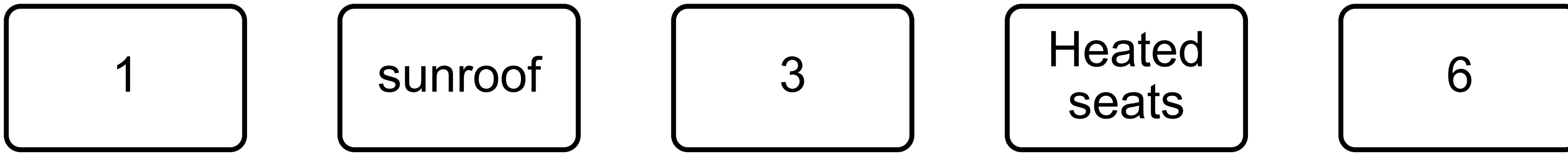
Car Factory Assembly Line—back to the days of Henry Ford

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

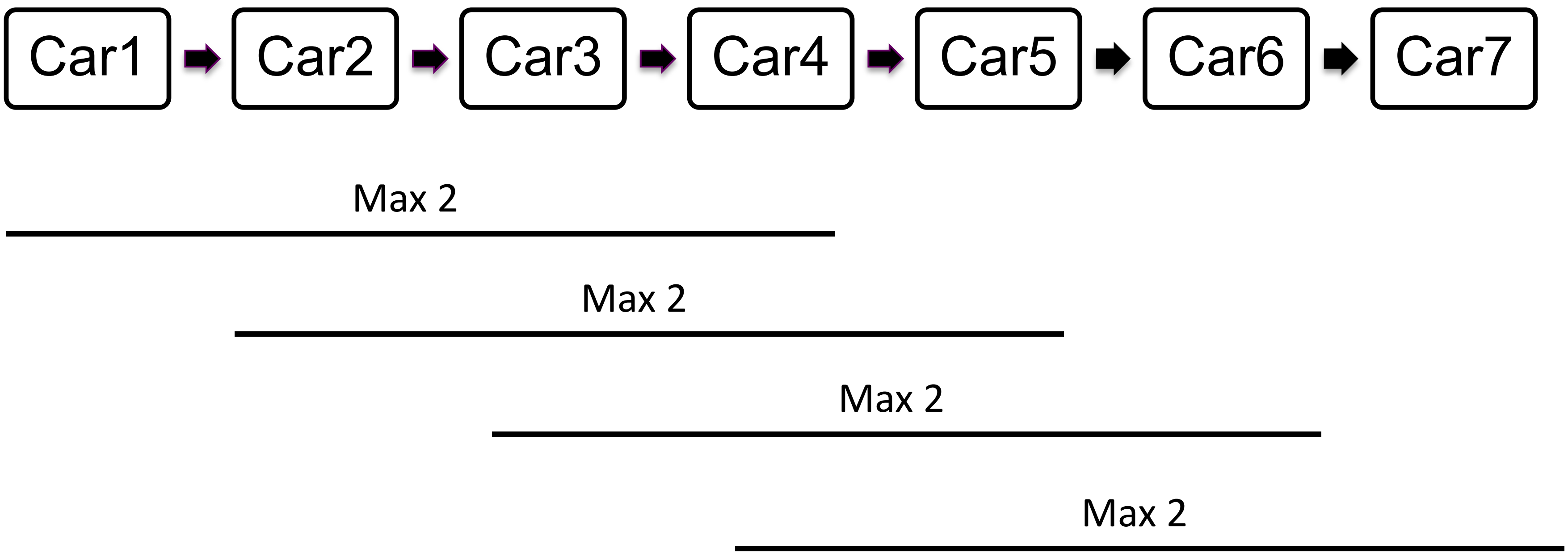
Slot1 → Slot2 → Slot3 → Slot4 → Slot5 → Slot6 → Slot7

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

Car1 → Car2 → Car3 → Car4 → Car5 → Car6 → Car7

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 $C_1, \ldots, C_m$.

Formalization of a CSP

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

- Each constraint C
  - Constrains a particular 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
      - e.g., C(V1=a, V2=b, V4=c) → True
    - False—this assignment falsifies the constraint.
      - e.g., C(V1=a, V2=b, V4=a) → False

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

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

- Solutions:
  - 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.

Example: Sudoku

- Variables: V_{11}, V_{12}, …, V_{21}, V_{22}, …, V_{91}, …, V_{99}
- Domains:
  - Dom[V_{ij}] = {1-9} for empty cells
  - Dom[V_{ij}] = {k} a fixed value k for filled cells.

V_{33} =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

V_{88} = 3

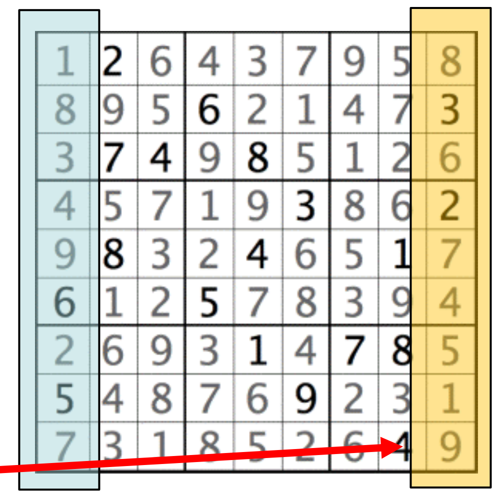
Example: Sudoku

- Variables: V_{11}, V_{12}, …, V_{21}, V_{22}, …, V_{91}, …, 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}, …, V_{19})
    - All-Diff(V_{21}, V_{22}, V_{23}, …, V_{29})
    - …., All-Diff(V_{91}, V_{92}, …, V_{99})

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

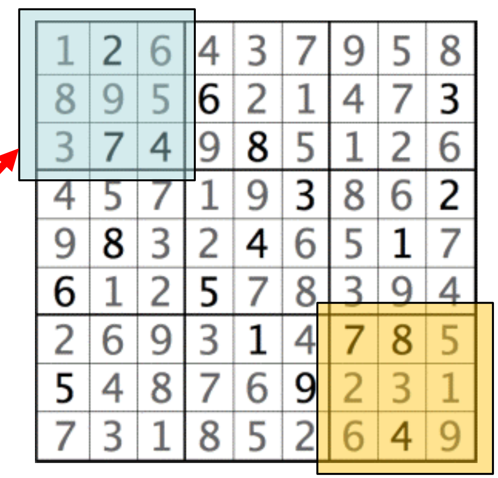
Example: Sudoku

- Variables: V_{11}, V_{12}, …, V_{21}, V_{22}, …, V_{91}, …, V_{99}
- Domains:
  - $\mathrm{Dom}[V_{ij}]$ = {1-9} for empty cells
  - $\mathrm{Dom}[V_{ij}]$ = {k} a fixed value k for filled cells.
- Constraints:
  - Row constraints:
    - All-Diff(V_{11}, V_{12}, V_{13}, …, V_{19})
    - All-Diff(V_{21}, V_{22}, V_{23}, …, V_{29})
    - …., All-Diff(V_{91}, V_{92}, …, V_{99})
  - Column Constraints:
    - All-Diff(V_{11}, V_{21}, V_{31}, …, V_{91})
    - All-Diff(V_{21}, V_{22}, V_{13}, …, V_{92})
    - …., All-Diff(V_{19}, V_{29}, …, V_{99})



Example: Sudoku

- Variables: V_{11}, V_{12}, …, V_{21}, V_{22}, …, V_{91}, …, 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}, …, V_{19})
    - All-Diff(V_{21}, V_{22}, V_{23}, …, V_{29})
    - …., All-Diff(V_{91}, V_{92}, …, V_{99})
  - Column Constraints:
    - All-Diff(V_{11}, V_{21}, V_{31}, …, V_{91})
    - All-Diff(V_{21}, V_{22}, V_{13}, …, V_{92})
    - …., All-Diff(V_{19}, V_{29}, …, 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_{77}, V_{78}, V_{79},…, V_{97}, V_{98}, V_{99})



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.

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 satisfies the constraint.
  - Any assignment where two or more variables have the same value falsifies the constraint.

- This is a special kind of constraint called an ALL-DIFF constraint.
  - ALL-Diff(V1, .., Vn) can 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
    - But this collection of binary constraints has less pruning power under GAC (as we will see later)
    - ALL-DIFF appears in many CSP problems.

CSP as a Search Problem

A CSP could be viewed and solved as a traditional search problem

- However, CSPs do not require finding a path (to a goal). They only need the configuration of the goal state.

- Traditional search is an **inefficient** way to solve CSPs because it does not exploit the additional structure of the problem.
- This additional structure can be exploited in a process called constraint propagation.

Solving CSPs

- CSPs are best solved by a specialized version search called **Backtracking Search**.

- Key intuitions:
  - We can build up to a solution by searching through the space of partial assignments (rather than paths)
  - Order in which we assign the variables does not change the correctness of search – eventually they all have to be assigned. It can have a huge impact on the search's efficiency! We can decide on a suitable value for one variable at a time!
  - 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.

  ➔ This is the key idea of backtracking search.

Backtracking Search

- Specialized version of depth first search
- Explores partial assignments to the variables.
- A node is terminated if it violates a constraint
- A node specifying a total assignment is a solution.

Backtracking Search is one of the fundamental Computer Science Algorithms (Knuth "The Art of Computer Programming" Volume 4A is devoted to Backtracking Search)

Backtracking Search–Support Functions

- Class **Variable—**various member functions to deal with individual variables
  - **.domain()**
    Returns list of values in variable's domain
  - **.domainSize()**
    Returns number of values in domain
  - **.getValue() / .setValue()**
    Gets/sets variable's current assigned value. variable is **unassigned** if and only if its current assigned value is **None**
  - **.isAssigned()**
    Returns **True/False** if variable has an assigned value (i.e., its getValue() != **None)**
  - **.name()**
    Returns string specifying the variable's name (for documenting the CSP model)

Backtracking Search–Support Functions

- Class **Constraint—**various member functions to deal with individual constraints
  - **.scope()**
    Returns list of **variables** in the constraint's scope.
  - **.arity()**
    Returns number of variables in constraint's scope.
  - **.numUnassigned()**
    Returns number of variables in constraint's scope that are **not assigned**. (Other variables have been assigned)
  - .**check()**
    Returns **True** if the values currently assigned to the variables in **.scope**() satisfy the constraint. **False** if not.
    If **.numUnassigned() > 0** returns **True** (not yet able to check that the constraint is falsified).
  - **name()**
    Returns string specifying the constraint's name

Backtracking Search–Support Functions

- **variables**
  List containing all variables of the CSP problem (list of **Variable** class instances)
- **constraints**
  List containing all constraints of the CSP problem (list of **Contraint** class instances)
- **constraintsOf(var)**
  Returns list of constraints that have **var** in their scope (list of **Constraint** class instances)
- **allSolutions**
  Boolean flag. If **True** we want to enumerate all solutions

Backtracking Search–Support Functions

- We will extend these support function/variables when we discuss extensions of simple backtracking that do domain filtering (pruning)

Backtracking Search: The Algorithm BT

```
BT(unAssignedVars)     #pass collection of unassigned variables
  if unAssignedVars.empty():   #no more unassigned variables
    for var in variables:
      print var.name(), " = ", var.getValue()
    if allSolutions:
      return          #continue search to print all solutions
    else: EXIT        #terminate after one solution found.
  var := unAssignedVars.extract() #select next variable to assign
  for val in var.domain():
    var.setValue(val)
    constraintsOK = True
    for constraint in constraintsOf(var):
      if constraint.numUnassigned() == 0:
        if not constraint.check():
          constraintsOK = False
          break
    if constraintsOK:
      BT(unAssignedVars)
  var.setValue(None)           #undo assignemnt to var
  unAssignedVars.insert(var)   #restore var to unAssignedVars
  return
```

Backtracking Search—initial call

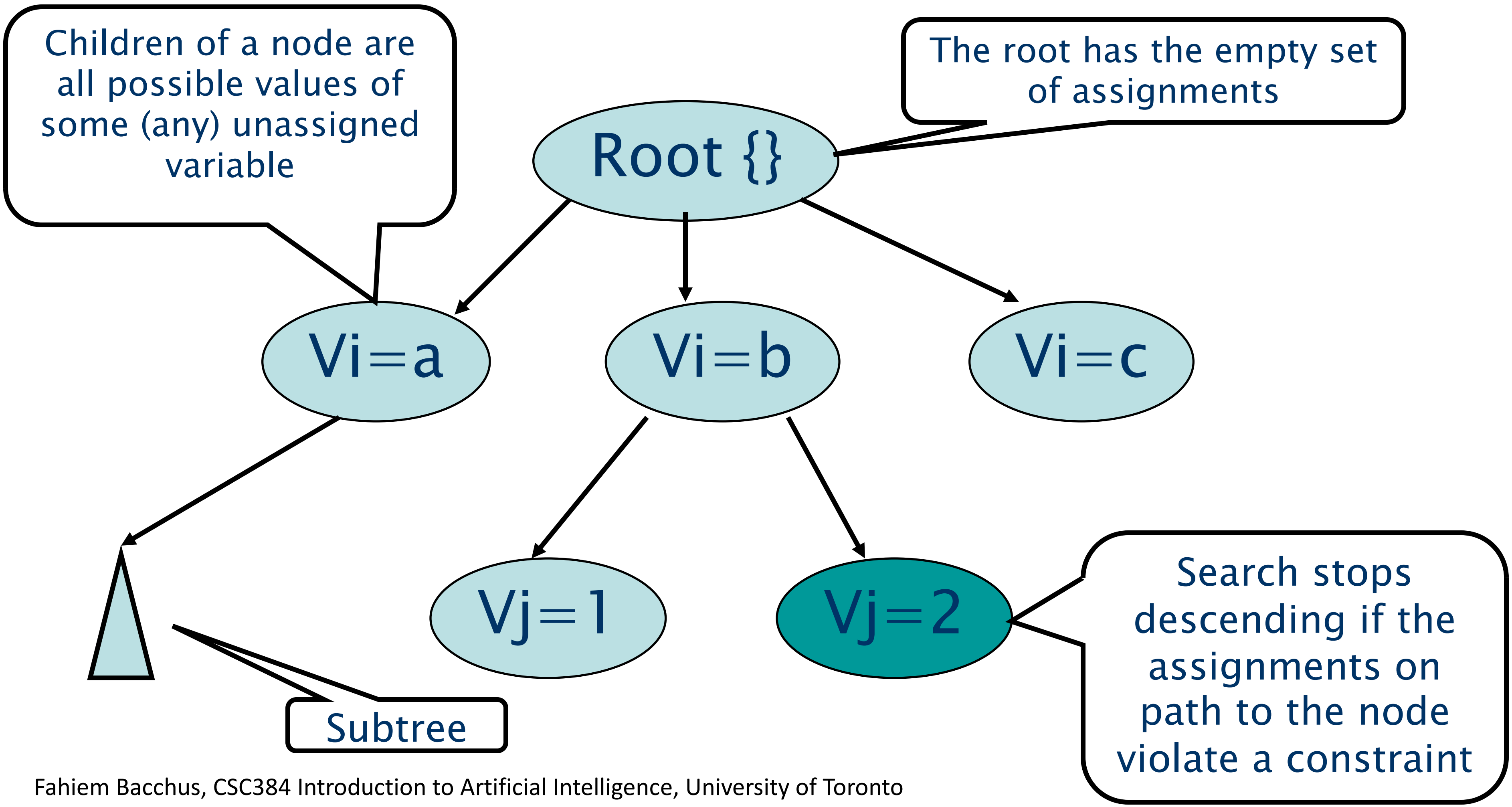
- Initially all variables are unassigned (var.getValueI() == None for every variable).
- Initially `unAssignedVars` contains all variables.

Backtracking Search–What variable to assign next?

- The decision of which variable to assign next does not affect whether or not backtracking search will find a solution.
- **But** it does have a tremendous impact on efficiency!

Backtracking Search

- The algorithm searches a tree of partial assignments.



Backtracking Search

- Heuristics are used to determine
  - the order in which variables are assigned: `UnAssignedVars.extract()`
  - determines the order of values tried for each variable.
  - As in search, this collection can be kept in heuristic order so that we always select the first variable.
  - **But** the best variable can change every time we assign another variable.
  - In general, the best variable depends on the collection of assigned variables as well as the values these variables have been assigned.

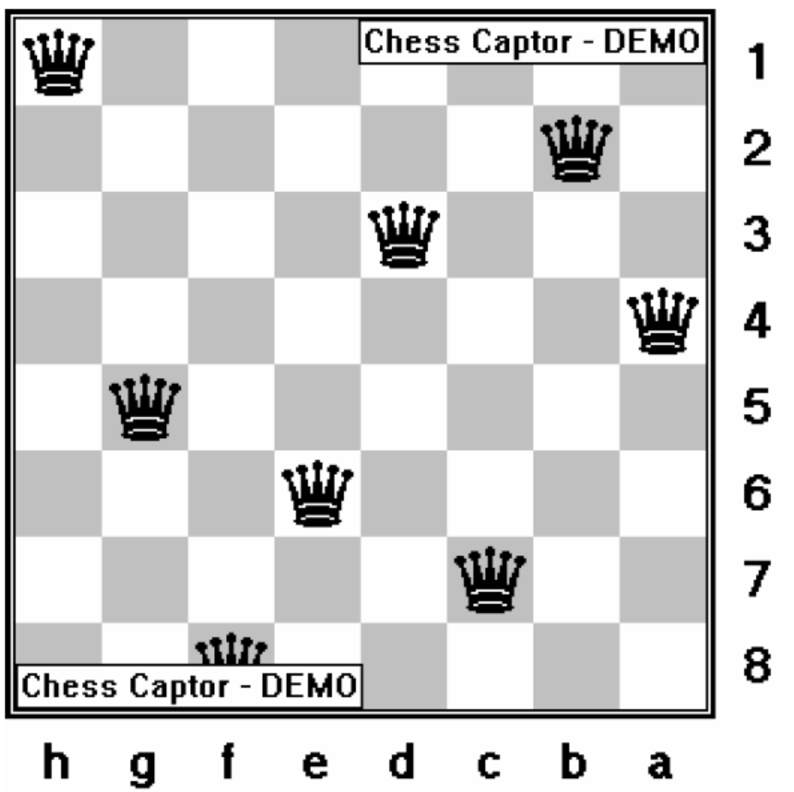
Backtracking Search

- Hence, 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 can have 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.

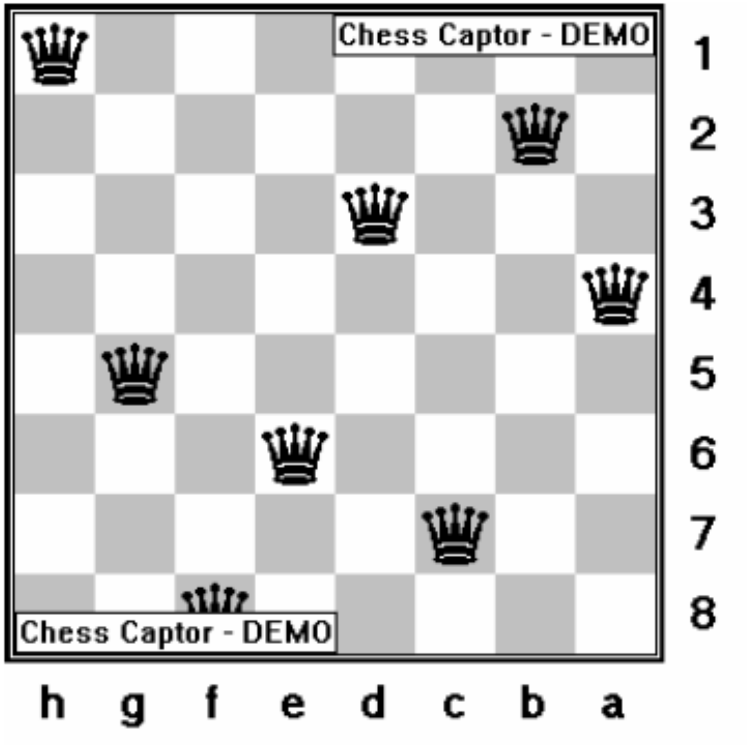


Example: N-Queens

- 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

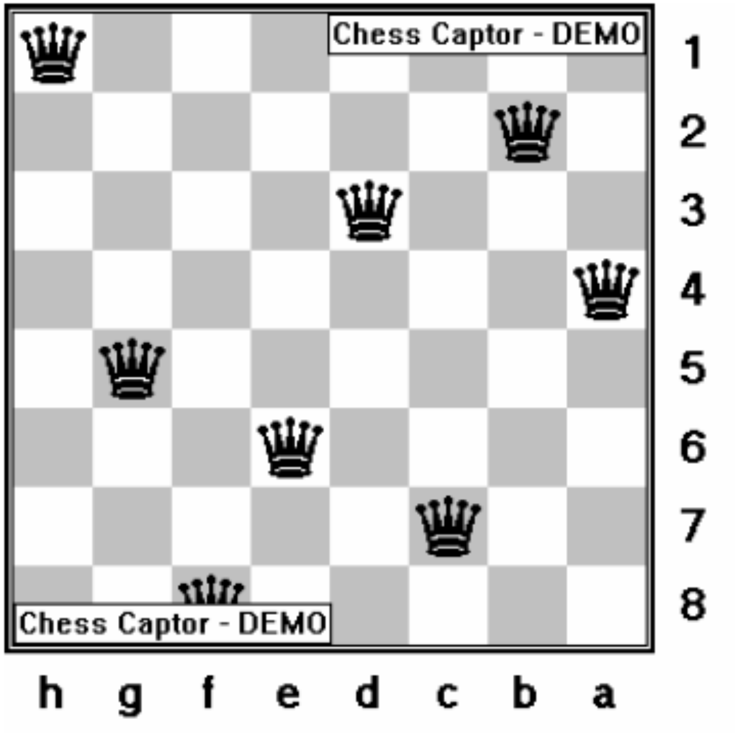
- Better Modeling:
  - N variables Qi, one per row.
  - Value of Qi is the column the Queen in row i is placed; possible values {1, …, N}.

- 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



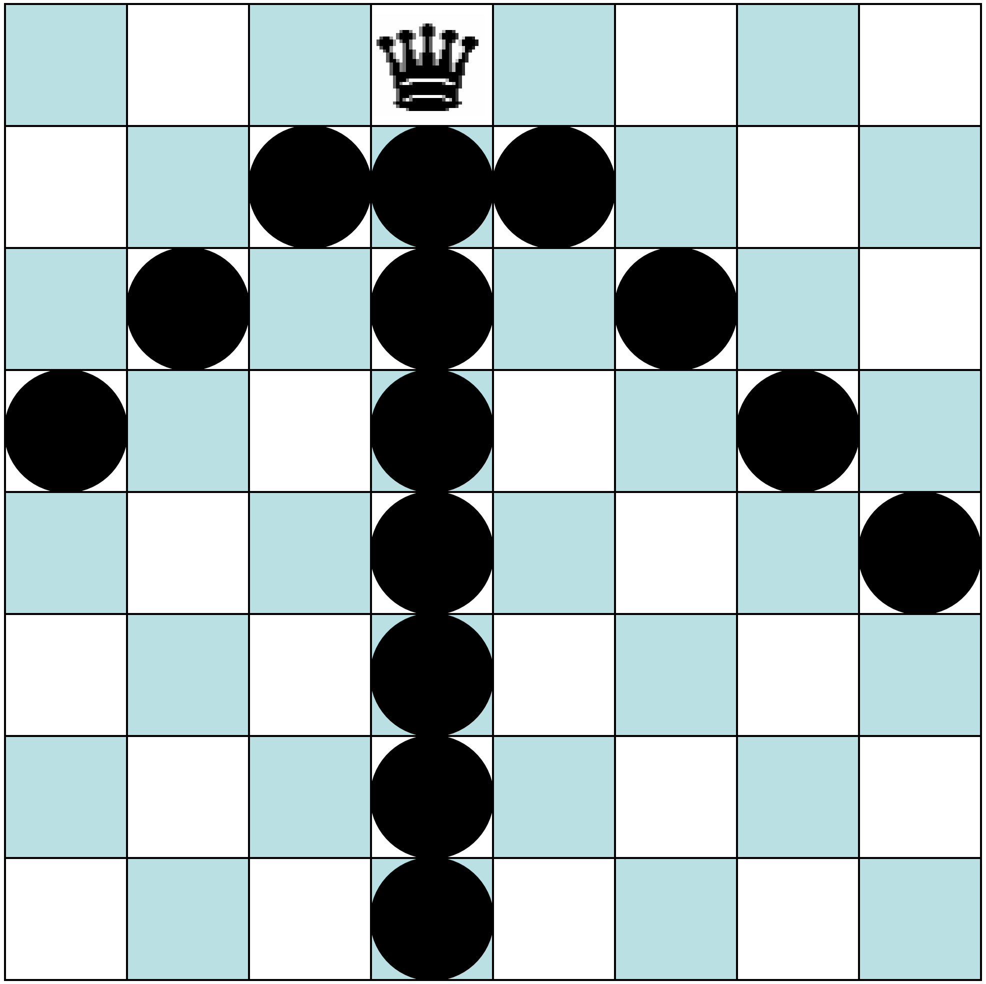
Example: N-Queens

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

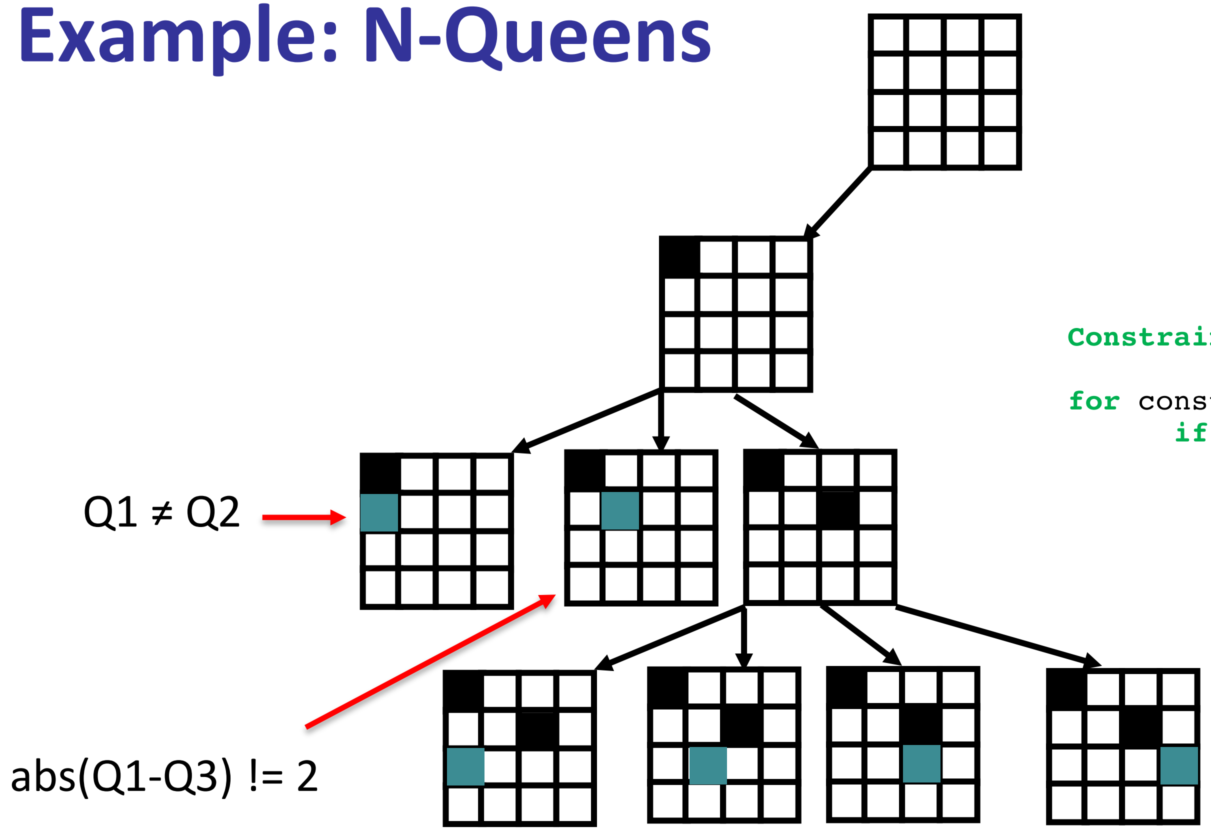
    $Qi \neq Qj$ for all $i \neq j$
  - Diagonal constraints

    $abs(Qi-Qj) \neq abs(i-j)$
    - i.e., the difference in the values assigned to Qi and Qj can't be equal to the difference between i and j.
- E.g. 4 Queens:
  $Q1 \neq Q2$, $Q1 \neq Q3$, $Q1 \neq Q4$, $Q2 \neq Q3$, $Q2 \neq Q4$, $Q3 \neq Q4$,
  abs(Q1-Q2) != 1, abs(Q1-Q3) != 2, abs(Q1-Q4) != 3, abs(Q2-Q3) != 1, abs(Q2-Q4) != 2, abs(Q3-Q4) != 1

Example: N-Queens



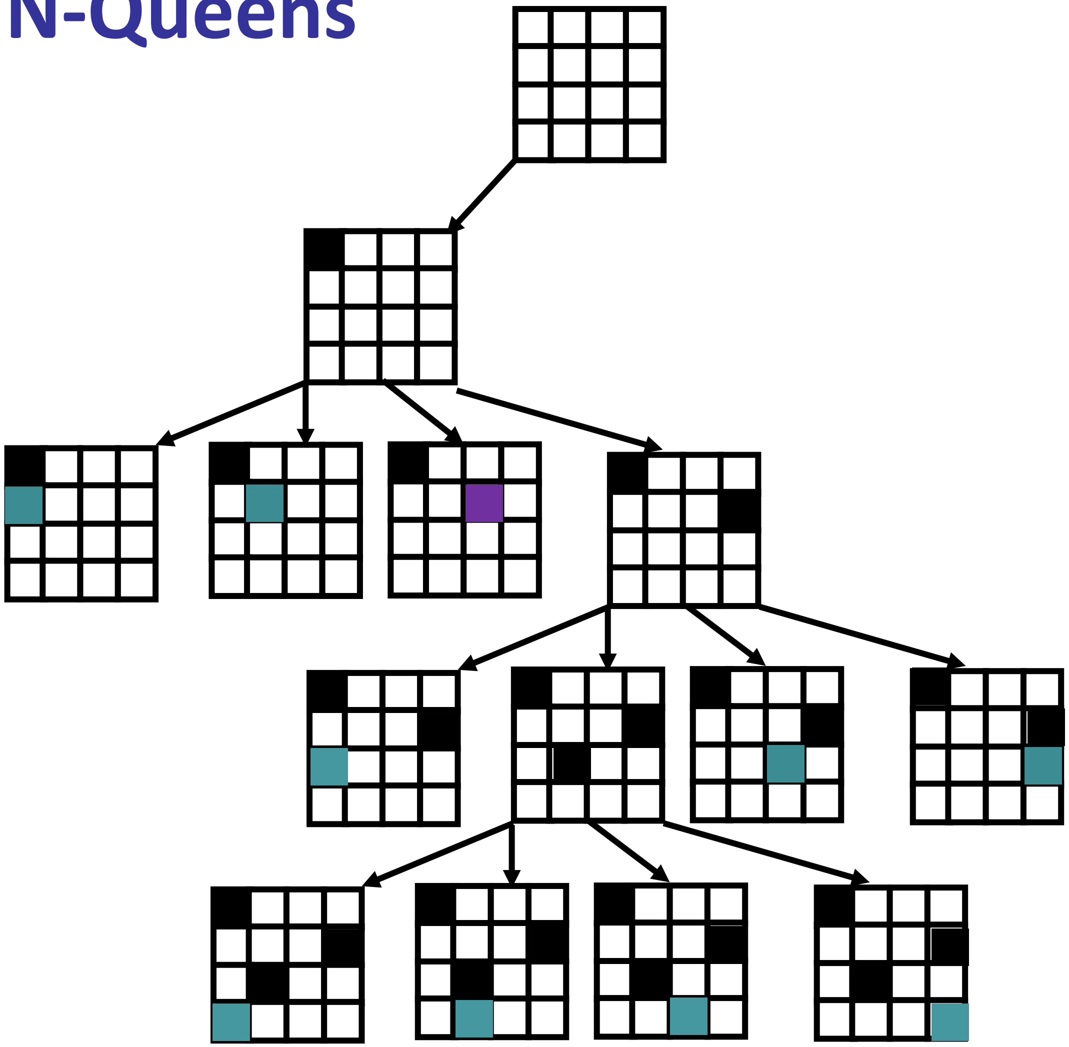
Example: N-Queens



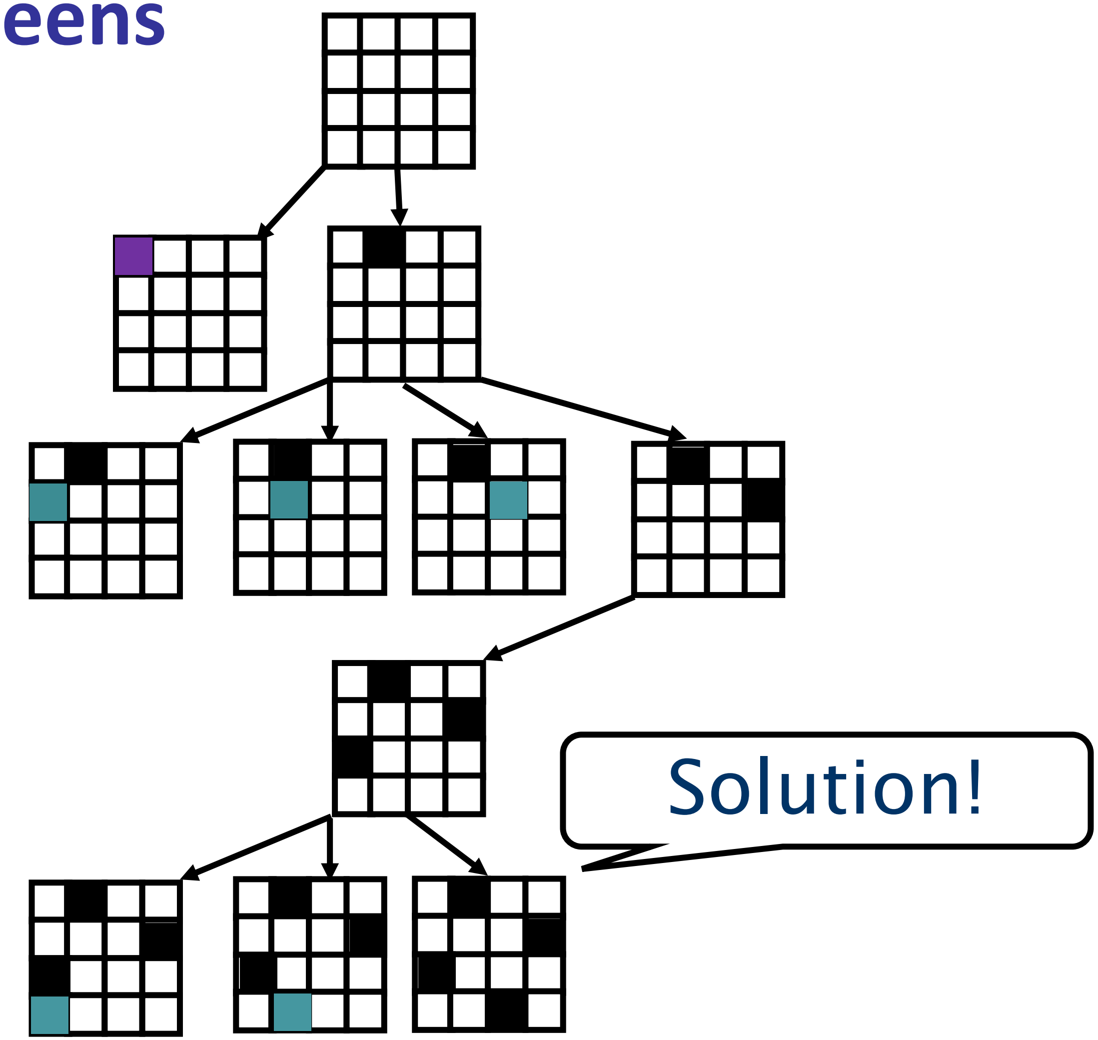
Constraint Check inner loop of BT:

```
for constraint in constraintsOf(var):
      if constraint.numUnassigned() == 0:
        if not constraint.check():
          constraintsOK = False
          break
```

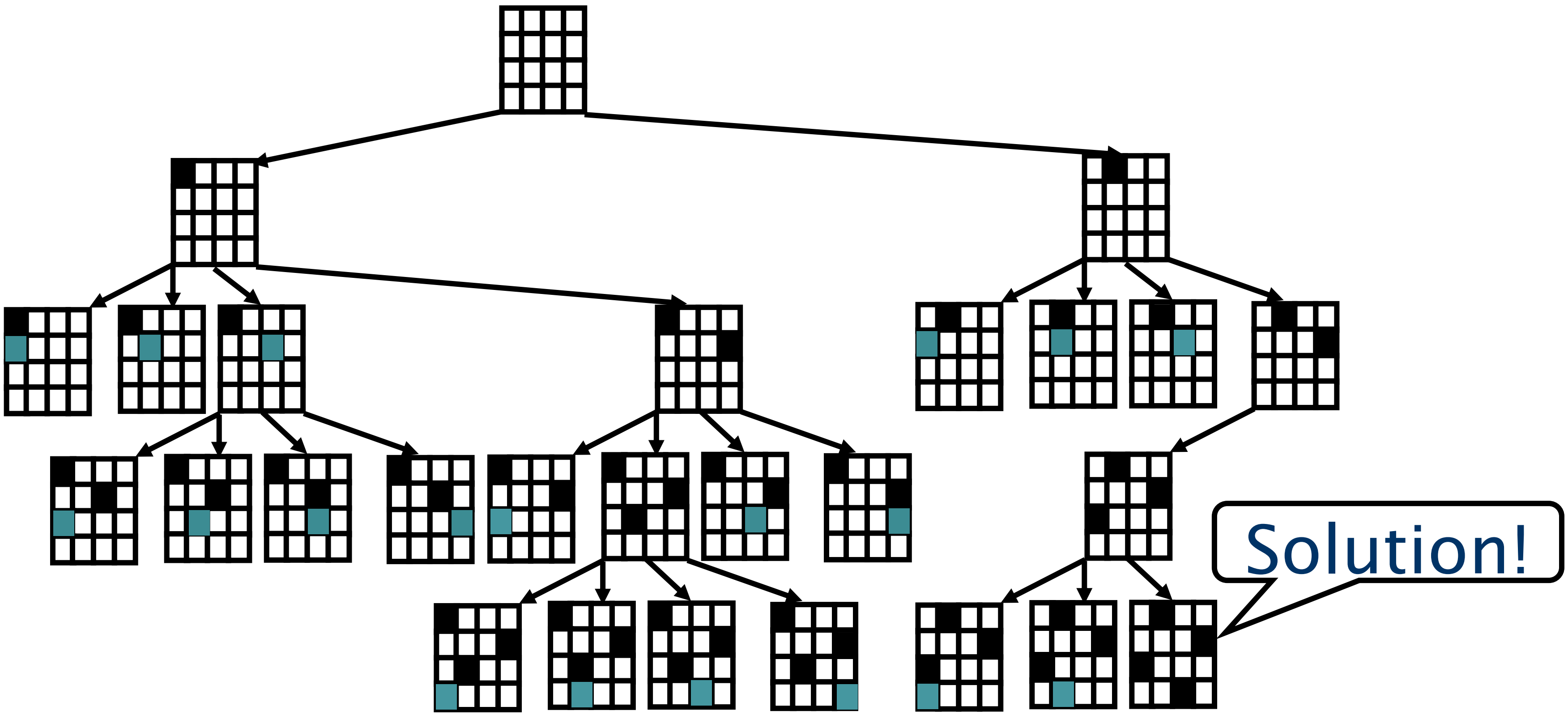
Example: N-Queens



Example: N-Queens



Example: N-Queens



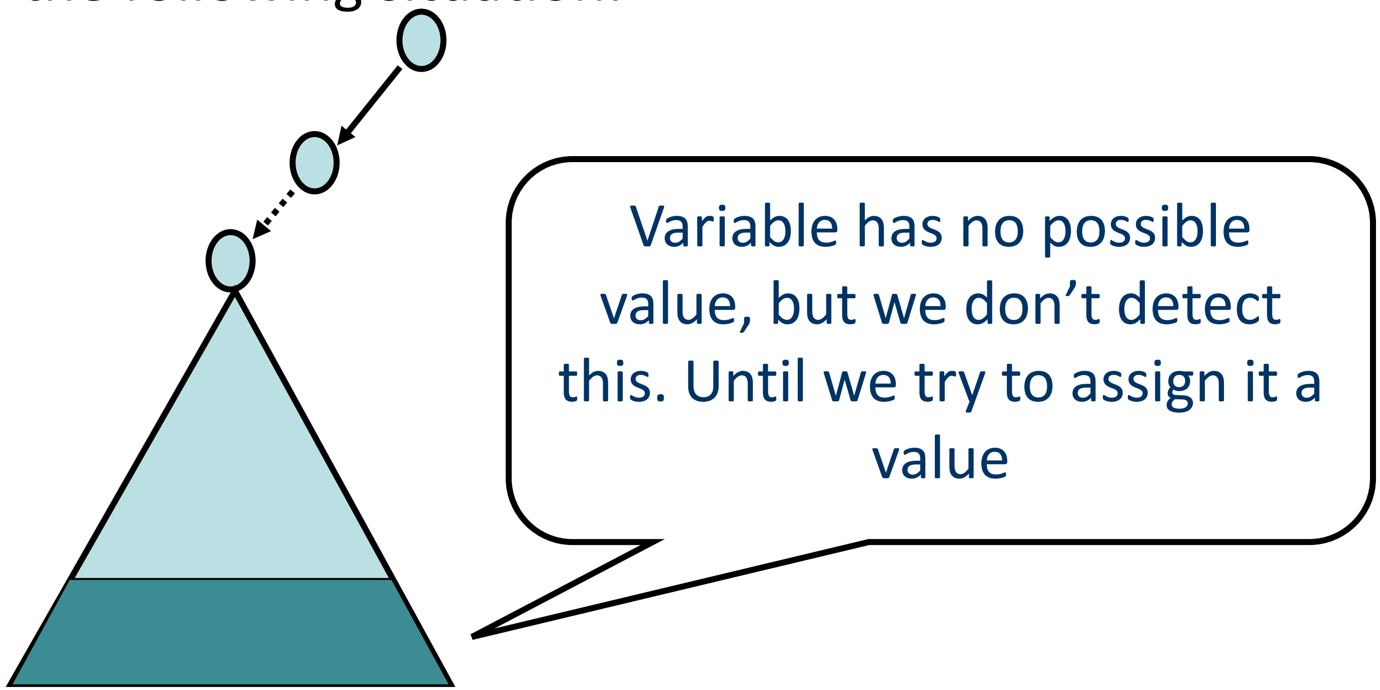
Problems with Plain Backtracking

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:



- Leads to the idea of constraint propagation/domain filtering

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, it requires 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.
  - A similar tradeoff occurs if we try to compute an expensive heuristic in A* search—we might expand fewer nodes but take more time.

- 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 (constraint.numUnassigned() = 1)

- For that uninstantiated variable, we check all of its values, pruning those values that violate the constraint.

Forward Checking–Support Functions

- During search the domains of unassigned variables (future variables) can be pruned because these values are incompatible with the values given to currently assigned variables.
- During backtracking search, we will be making new variable assignments, and undoing them when we backtrack.

Forward Checking–Support Functions

- Hence an important component of algorithms that use constraint propagation is that
  1. They must have facilities to prune values from the domains of the future variables—we have to **keep track of the unpruned values** remaining for the future variables.
  2. They must also have the facility to **restore** pruned values to the domains of the future variables on backtrack when we undo some variable assignments.
     - This is accomplished by keeping track of which variable assignment caused the pruning to occur, and then restoring the values pruned by that variable assignment when it is undone by backtracking

Forward Checking–Support Functions

- Additional member functions/variables for **Variable class**
  - **.curDomain()**
    Returns list of variable's current values (the currently unpruned values).
  - **.curDomainSize()**
    Returns number of variable's current values.
  - **.pruneValue(value, assigned_var, assigned_val)**
    Removes value from variable's current values. Remembers that **assigned_var** being assigned **assigned_val'** is the reason this value is being pruned

Forward Checking–Support Functions

- Additional member function for **Constraint class**
  - **.unassignedVars()**
    Returns list of unassigned variables in constraint's **.scope()** (variable is unassigned if its current value is None)
- Additional function
  - **restoreValues(variable, value)**
    returns all values pruned because of the passed variable/value assignment to the current domain of their respective variable.

Forward Checking (detect values to prune)

- Called on a single constraint that has **numUnassigned()** = 1

```
FCCheck(constraint,assignedvar,assignedval):
#prune domain of unassigned variable
  var = constraint.unassignedVars()[0]
  #.unassignedVars() should be list of length 1
  #var is the single unassigned variable of constraint
  for val in var.curDomain():
    var.setValue(val)  #trial assign and check constraint
    if not constraint.check():
      var.pruneValue(val,assignedvar,assignedval)
  if var.curDomainSize() == 0:
     return "DWO" #domain wipe out
                  #no values left for var
  else: return "OK"
```

Forward Checking Algorithm

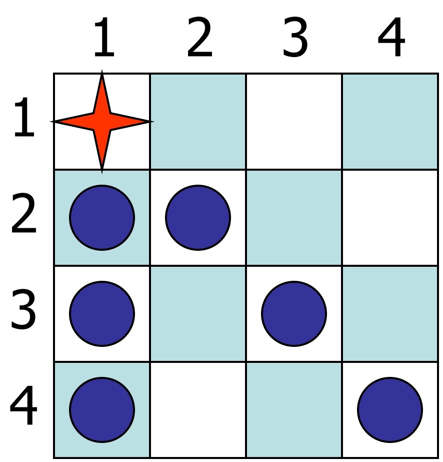
```
FC(unAssignedVars):   #Forward checking, pass unassigned variables
  if unassignedVars.empty():  #no more unassigned variables
    for var in variables:
      print var.name(), “,”, var.getValue()
      if allSolutions:
        return          #continue search to print all solutions
      else: EXIT      #terminate after one solution found.
  var = unassignedVars.extract() #select next variable to assign
  for val in var.curDomain():    #current domain!
    var.setValue(val)
    noDWO = True
    for constraint in constraintsOf(var):
      if constraint.numUnassigned() == 1:
        if FCCheck(C,var,val) == ”DWO” #prune future variables
          noDwo = False                  #pass var/val as reason
          break
    if noDwo:
      FC(unassignedVars)
    restoreValues(var, val)   #restore values pruned by this assigment
  var.setValue(None)          #undo assignemnt to var
  unAssignedVars.insert(var)  #restore var to unAssignedVars
  return
```

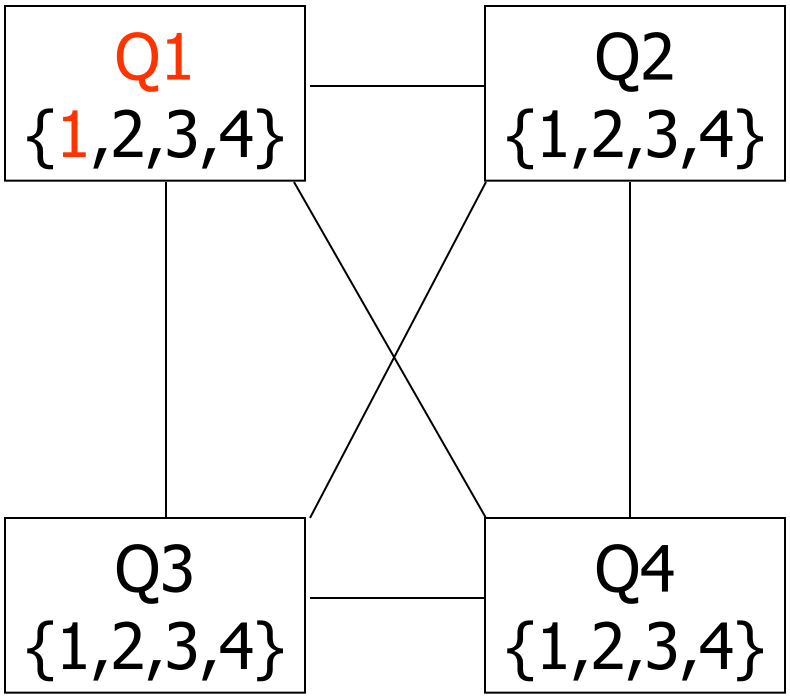
Forward Checking–initial call

- Initially all variables are unassigned
- Initially `unAssignedVars` contains all variables

4-Queens Problem

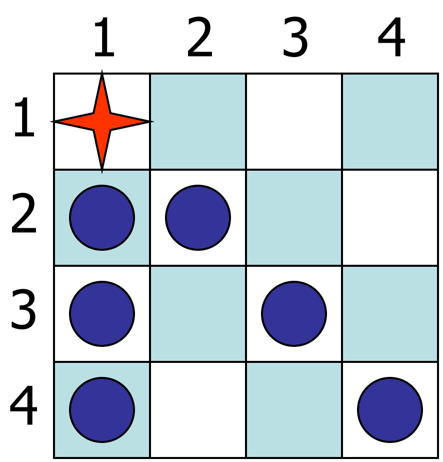
- Encoding with Q1, …, Q4 denoting a queen per row
  - cannot put two queens in same column. Binary constraint between every pair of variables.

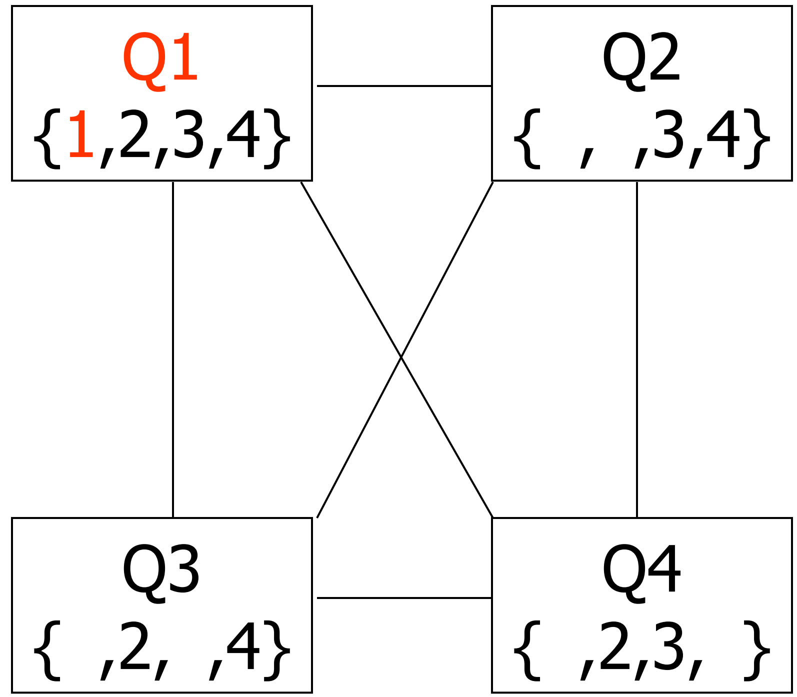




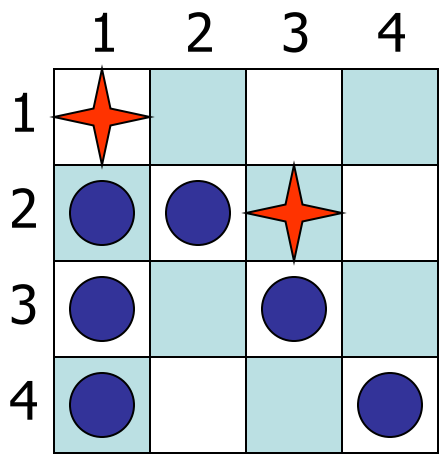
4-Queens Problem

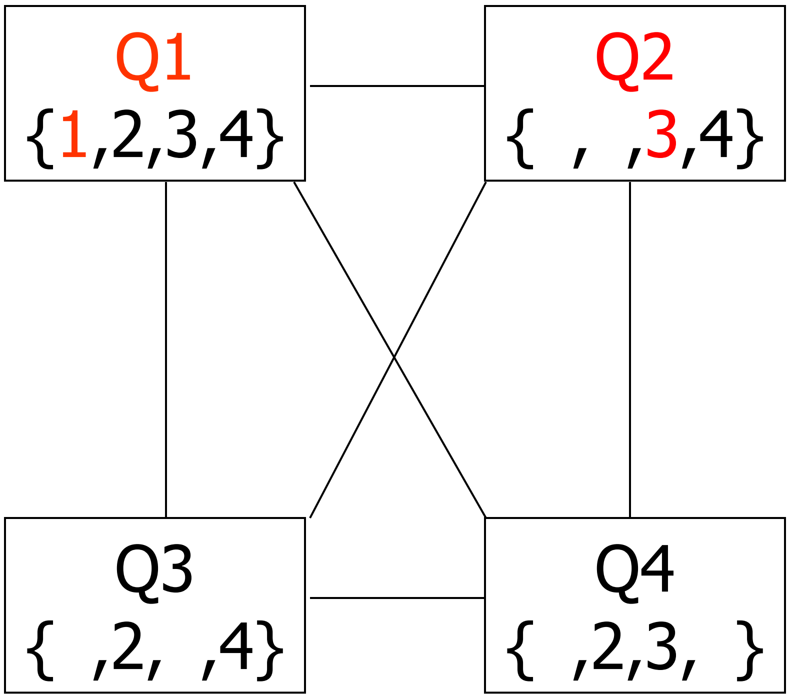
- Forward checking reduced the domains of all variables that are involved in a constraint with one unassigned variable:
  - each binary constraint with Q1



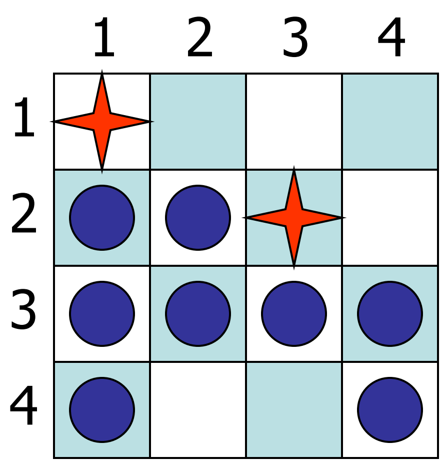


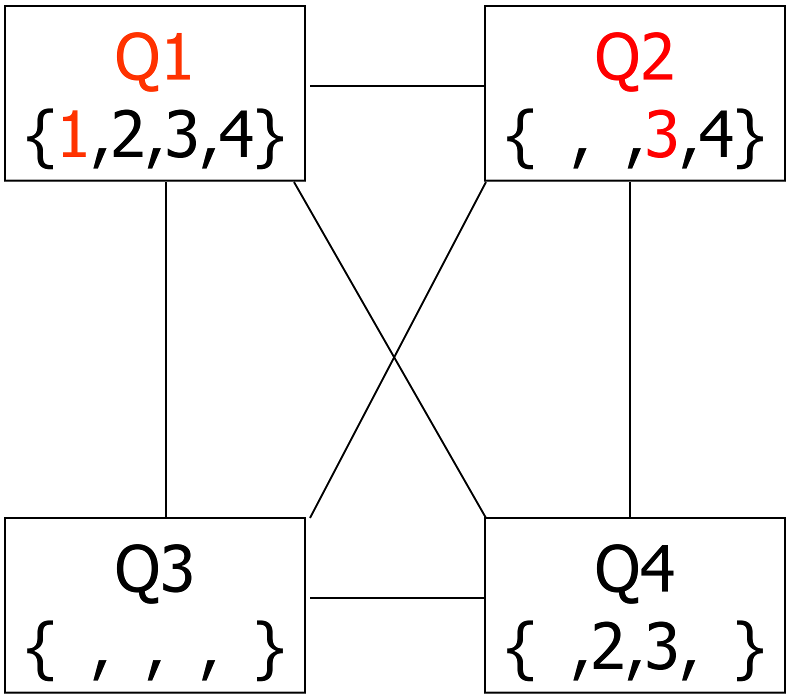
4-Queens Problem





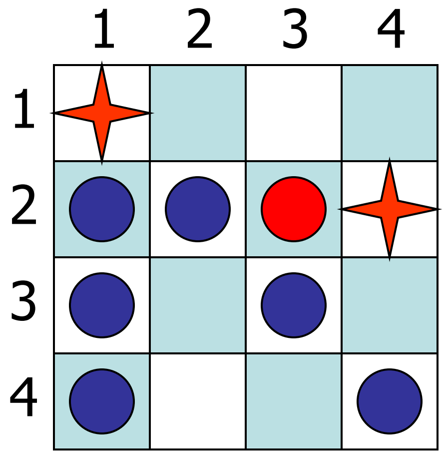
4-Queens Problem

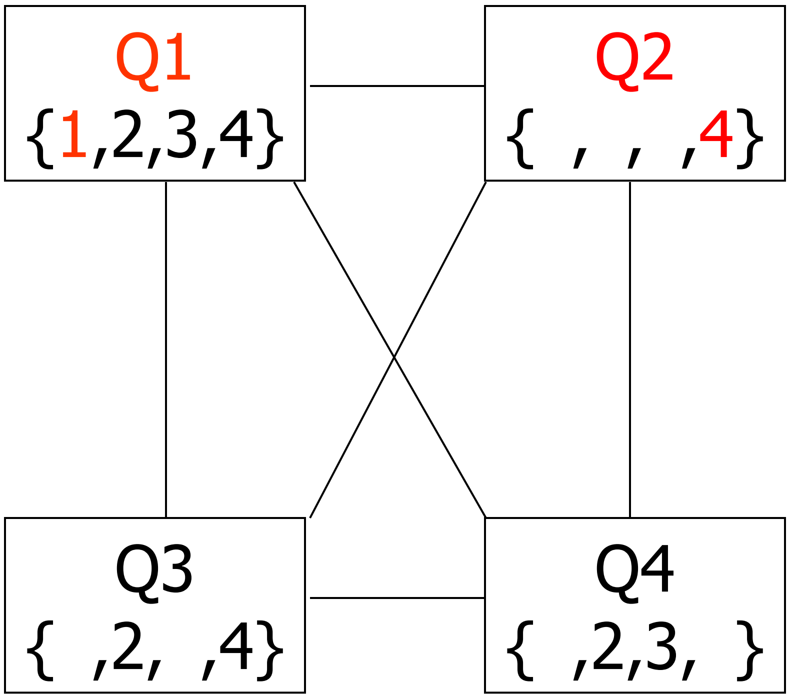




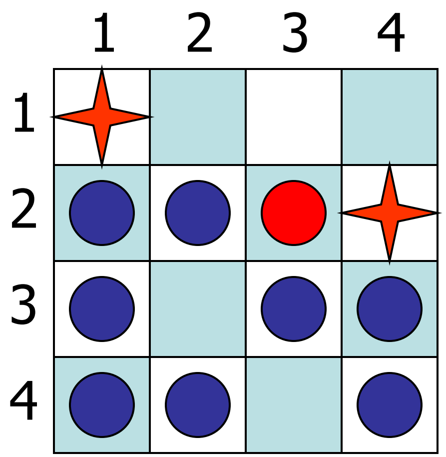
DWO

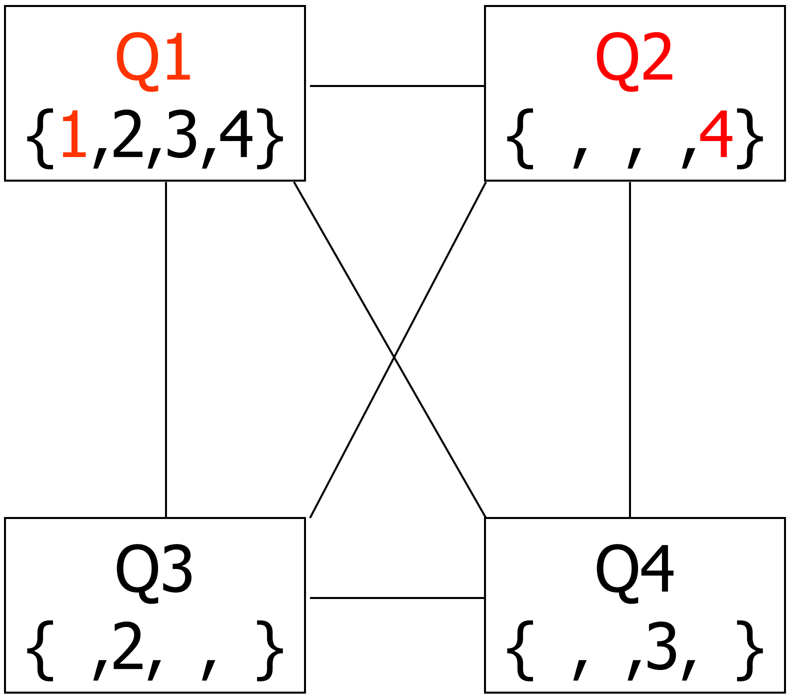
4-Queens Problem



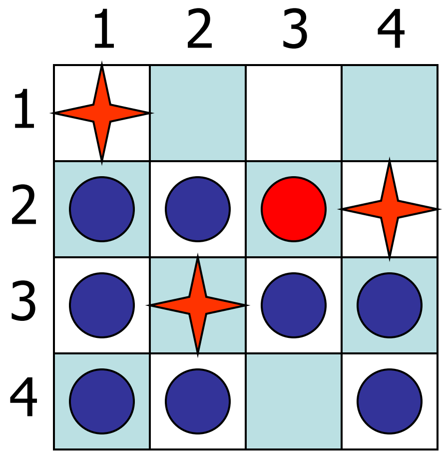


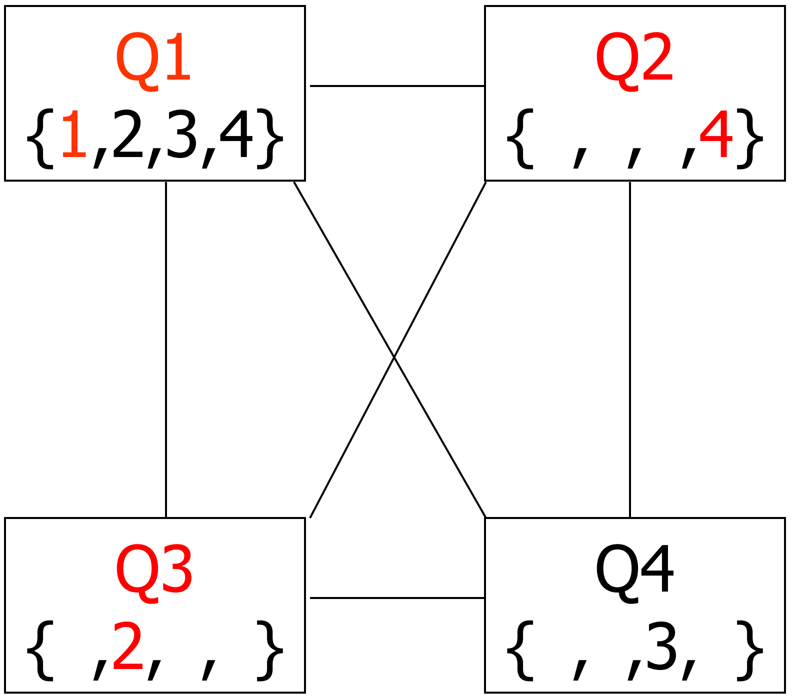
4-Queens Problem



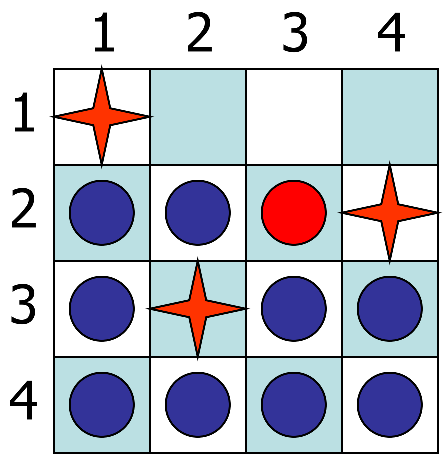


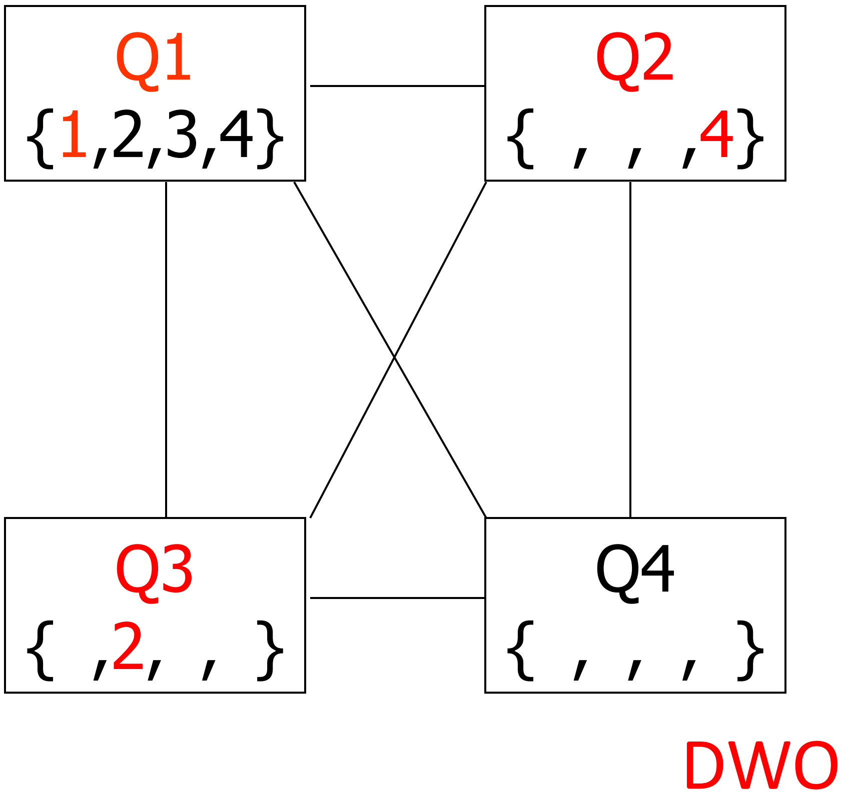
4-Queens Problem





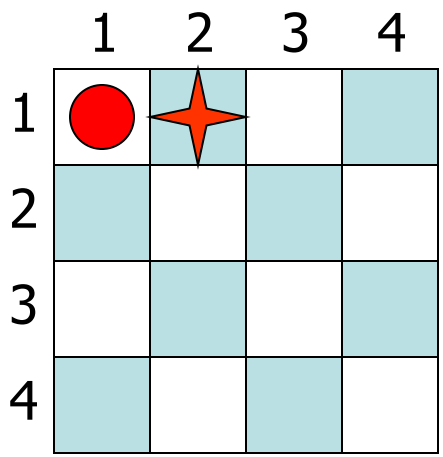
4-Queens Problem

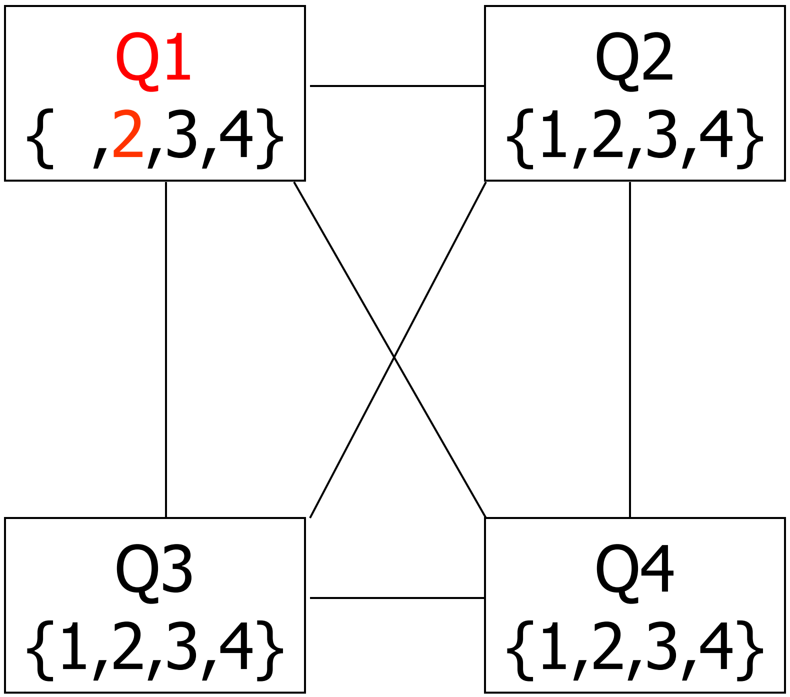




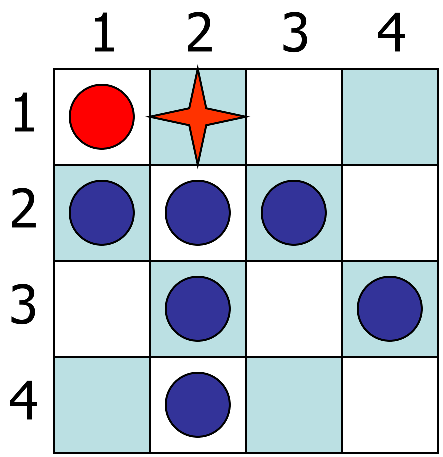
4-Queens Problem

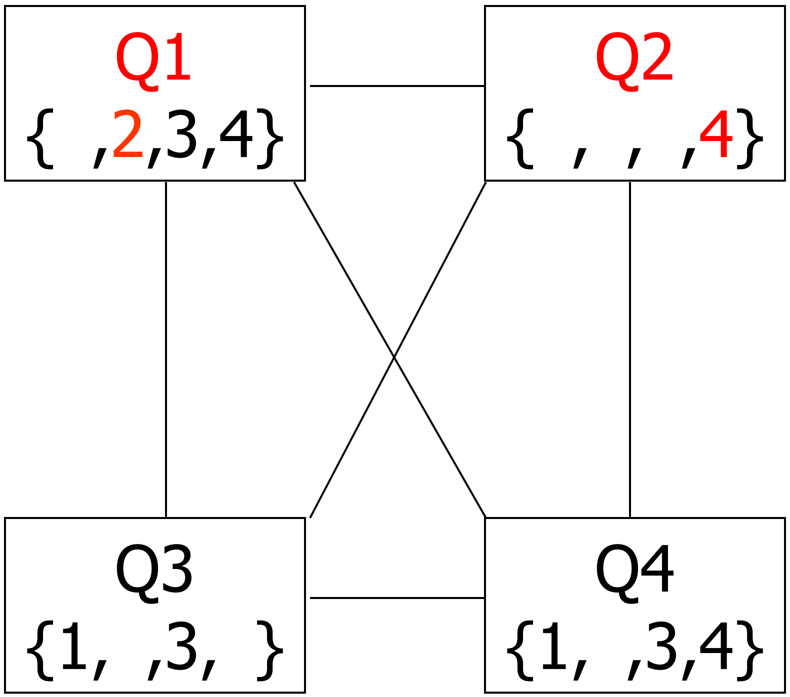
- Exhausted the subtree with Q1=1; try now Q1=2



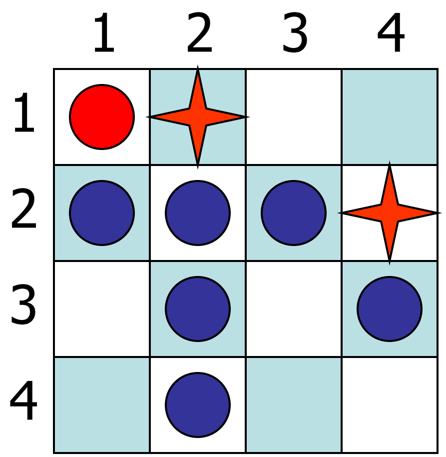


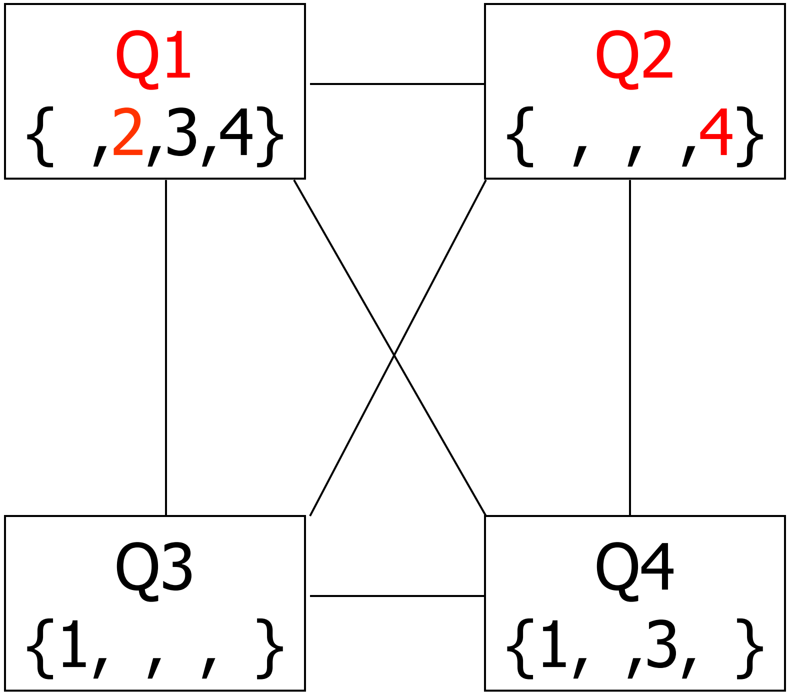
4-Queens Problem



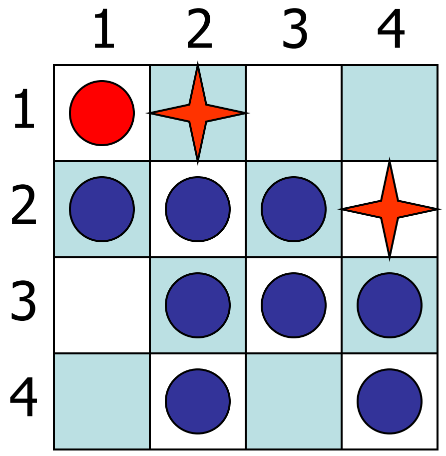


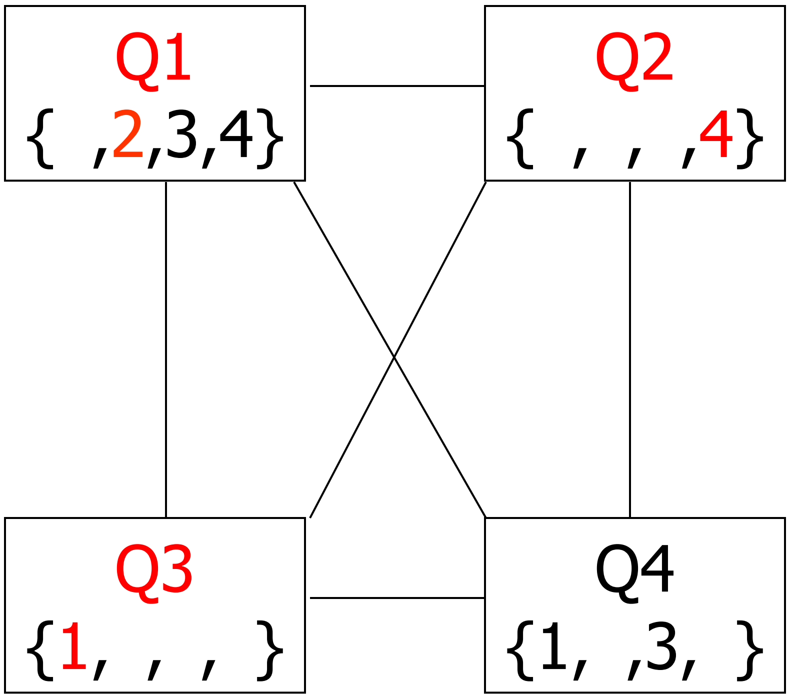
4-Queens Problem



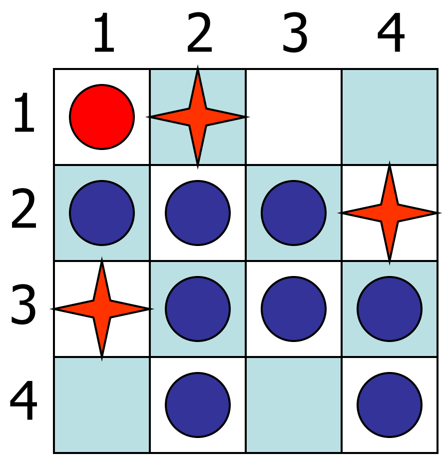


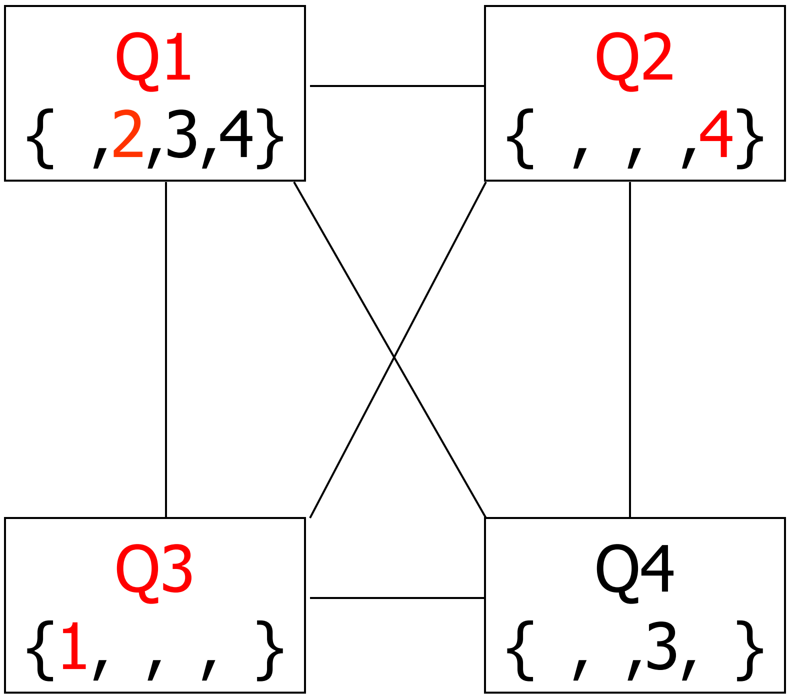
4-Queens Problem





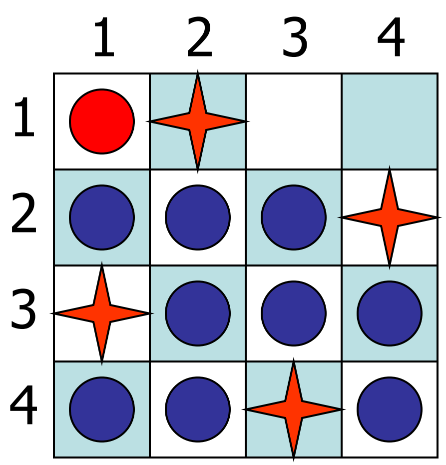
4-Queens Problem

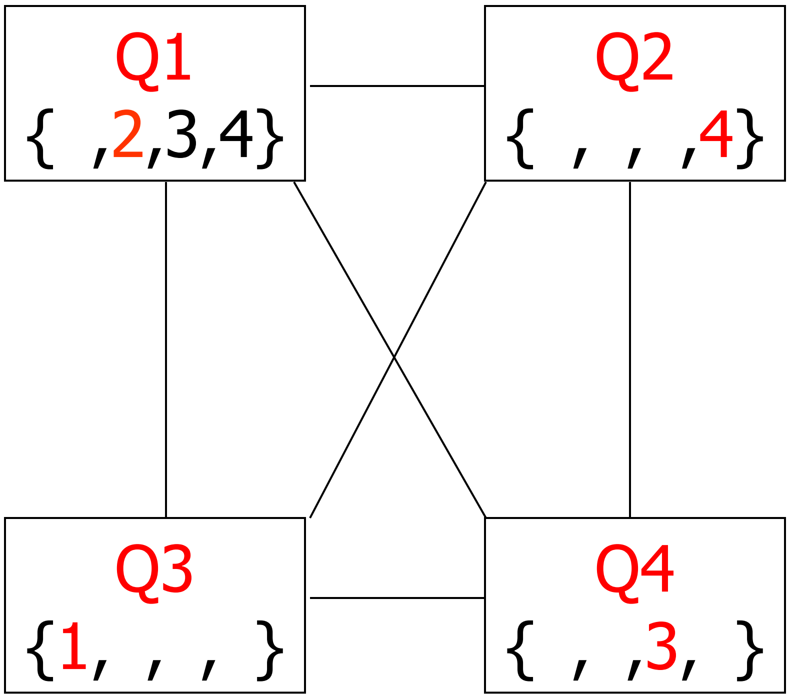




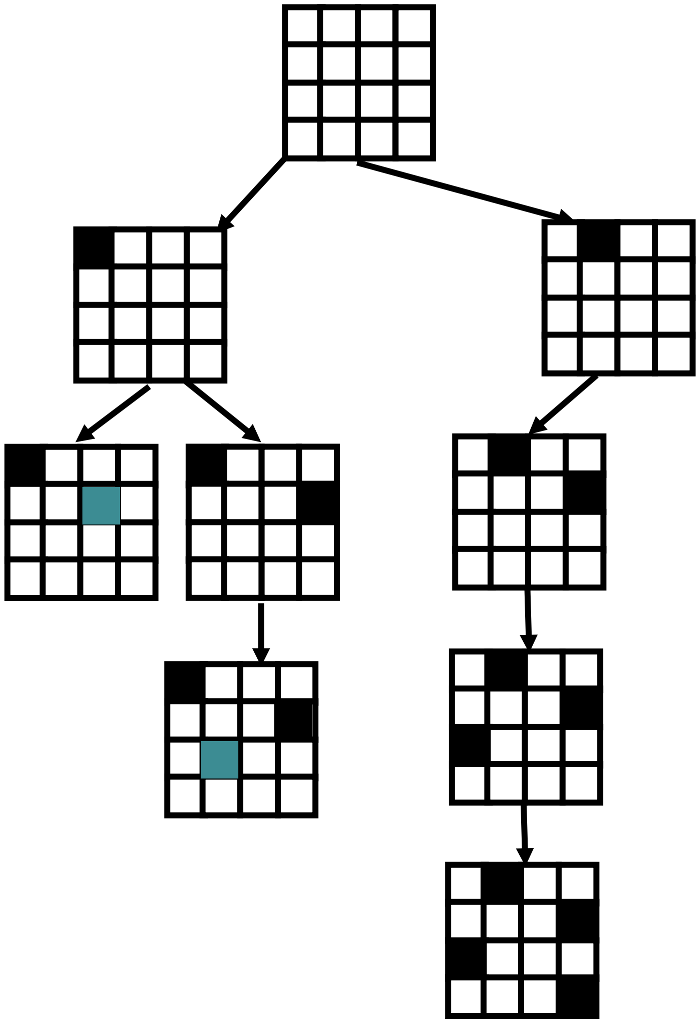
4-Queens Problem

- 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: 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 **.curDomainSize()**).
  - If a variable has only one value left, that value is forced, so we should assign it and propagate its consequences right away
  - 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

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}
→ Domain = {2}

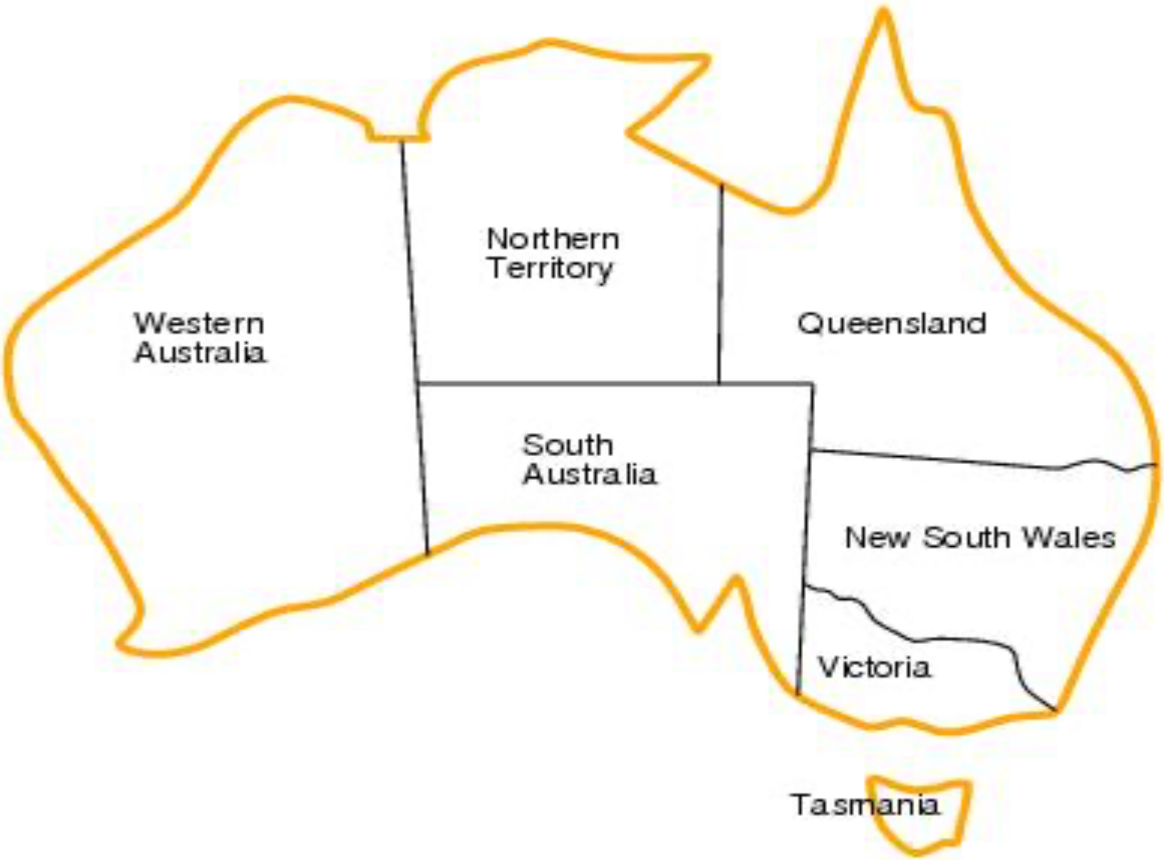
(9, 5): impossible values:
Row: {1, 5, 7, 8, 9}
Column: {1, 3, 4, 5, 7, 9}
Subsquare: {1, 5, 7, 9}
→Domain = {2, 6}

After assigning value 2 to cell (1,5): Domain of (9,5) = {6}

Most restricted variables! = MRV

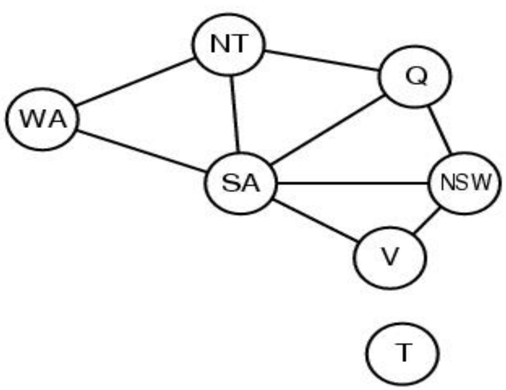
Example – Map Colouring

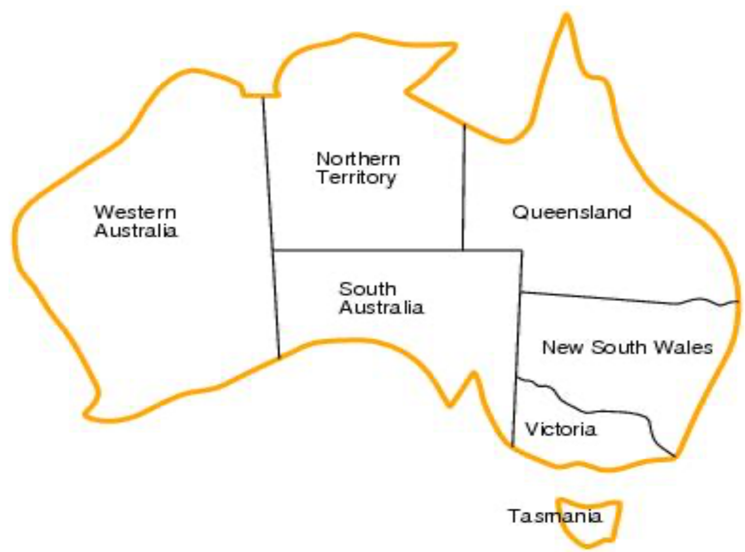
- Color the following map using red, green, and blue such that adjacent regions have different colors.



Example – Map Colouring

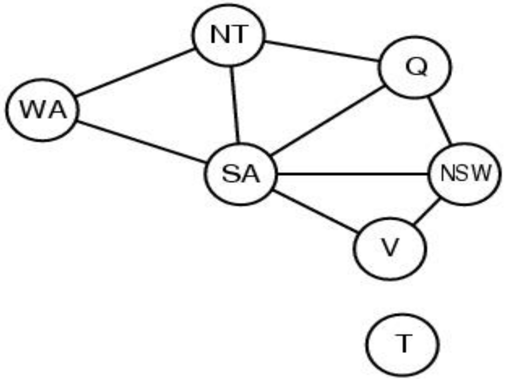
- Modeling
  - Variables: WA, NT, Q, NSW, V, SA, T
  - Domains: D_i={red, green, blue}
  - Constraints: adjacent regions must have different colors.
    - E.g. WA $\neq$ NT





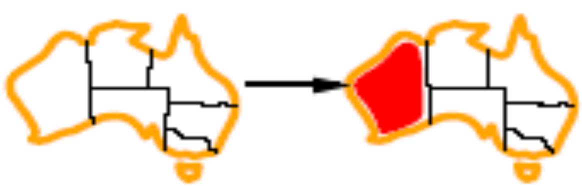
Example – Map Colouring

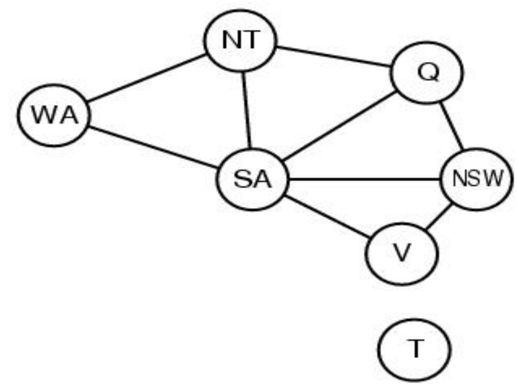
- Forward checking: keep track of remaining legal values for unassigned variables.
- Terminate search when any variable has no legal values.

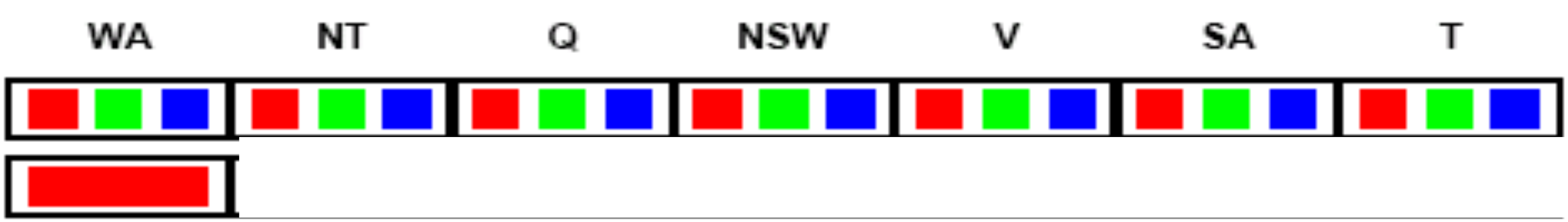


Example – Map Colouring

- Assign {WA=red}

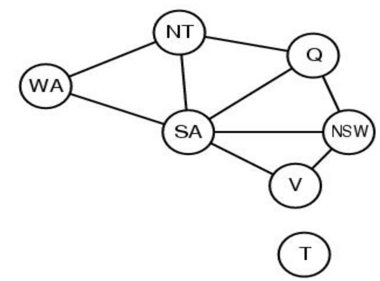






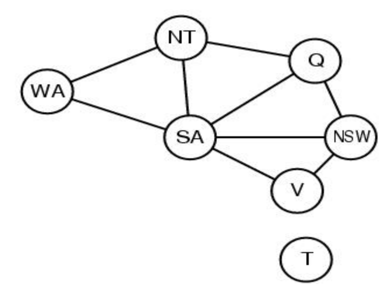
Example – Map Colouring

- Effects on other variables connected by constraints to WA
  - NT can no longer be red
  - SA can no longer be red
- Assign {Q=green} (Note: Not using MRV)



Example – Map Colouring

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

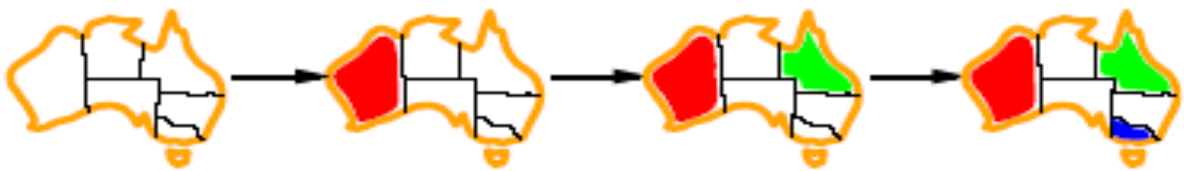


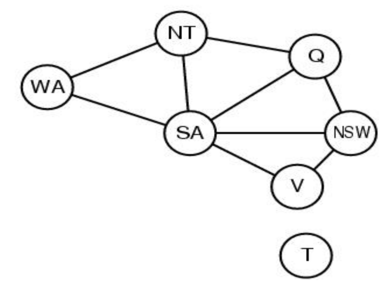
Example – Map Colouring

- 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.





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: Generalized Arc Consistency

- GAC—Generalized Arc Consistency is the most commonly used domain filtering (propagation) method used.
- Plain Backtracking check a constraint only when it has **.numUnassigned() = 0**
- Forward checking checks a constraint only when it has **.numUnassigned() = 1**
- **GAC** checks all constraints! This leads to much more pruning in general.
  - GAC ensures that all constraints satisfy a certain level of consistency (called GAC!) with respect to the already assigned variables
  - This level of consistency is achieved by pruning values from the domains of the unassigned variables.
  - Even at the root before any variables have been assigned, we can get some pruning by making the constraints GAC consistent.

Constraint Propagation: Generalized Arc Consistency

First we define formally the notion of consistency.

1. A **constraint**, $C(V_1, V_2, V_3, \ldots, V_n)$ 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.

Constraint Propagation: Generalized Arc Consistency

2. C(V1, V2, …, Vn) is GAC if and only if

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

3. A CSP is GAC if and only if

   all of its constraints are GAC.

   So we achieve GAC by achieving GAC for each constraint, and we achieve GAC for a constraint by achieving for every variable in the constraint's scope.

Constraint Propagation: GAC

- Achieving GAC for a constraint with respect to a particular variable:
  - Say we find a value d of variable Vi that is not consistent: That is, there is no assignments to the other variables that satisfy the constraint when Vi = 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).

- e.g. C(X,Y): X>Y Dom(X)={1,5,11} Dom(Y)={3,8,15}

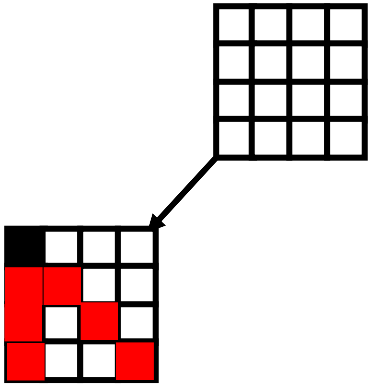
  - For X=1 there is no value of Y s.t. 1>Y => so we can remove 1 from domain X
  - For Y=15 there is no value of X s.t. X>15, so remove 15 from domain Y
  - We obtain the more restricted domains Dom(X)={5,11} and Dom(Y)={3,8}

Constraint Propagation: GAC

- **Propagation:** pruning the domain of a variable to make a constraint GAC can make a different constraint no longer GAC
- C1(X,Y): X>Y
  C2(Y,Z): Z<Y
  Dom(X)={1,5,11}, Dom(Y)={3,8,15}, Dom(Z) = {4,6}
- Make C1 GAC by pruning value 1 from Dom(X), and value 15 from
  Dom(X) = {5,11}
  Dom(Y) = {3,8}
- Make C2 GAC by pruning value 3 from Dom(Y)
  Dom(Y) = {8}
  Dom(Z) = {4, 6}
- **Now C1 is no longer GAC (if X=5 there is no value for Y)**

We need to re-achieve GAC for some constraints whenever a domain value is pruned.

Example: N-Queens GAC search Space

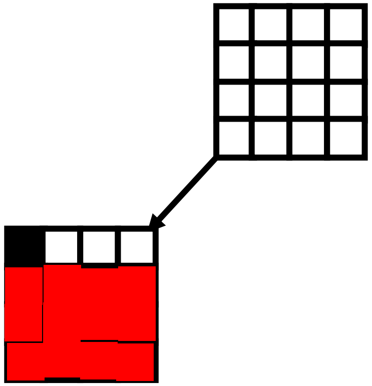


GAC immediately detects a DWO

1. When we assign V1 =1 we reduce Dom(V1) = {1}. V1's value is fixed.
2. Prune V2=1, V2=2, V3=1, V3=3, V4=1, V4=4—these values inconsistent with Dom(V1) = {1}
   Dom(V2) = {2,3}, Dom(V3) = {2,4}, Dom(V4) = {2,3}

Same as Forward Checking to this point

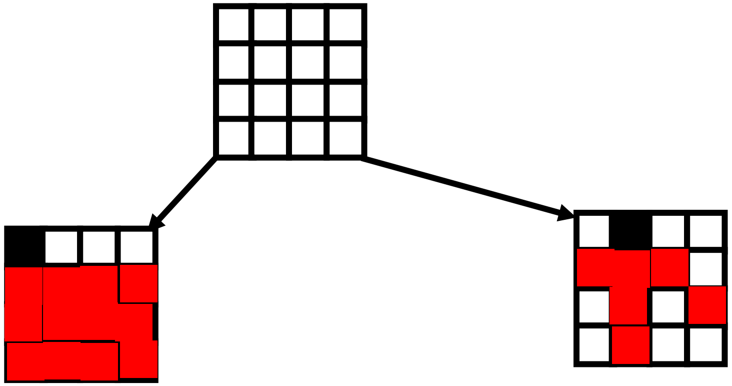
Example: N-Queens GAC search Space



GAC immediately detects a DWO

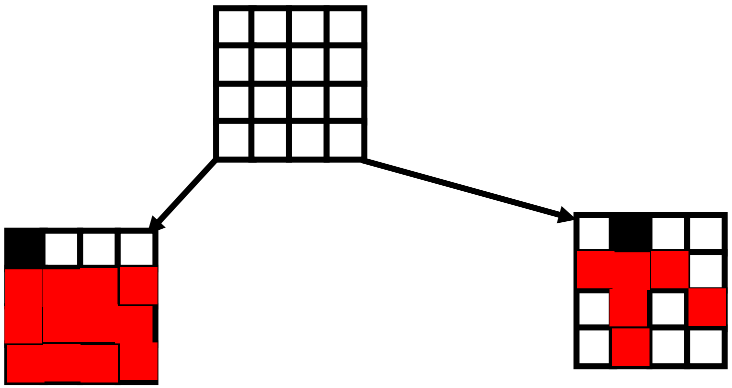
3. Dom(V2) = {3,4}, Dom(V3) = {2, 4}
   V2 = 3 has no compatible V3 value
   V3 = 4 has no compatible V2 value
   Dom(V2) = {4}, Dom(V3) = {2)
4. Dom(V3) = {2} , Dom(V4) = {2, 3}
   V3=2 has no compatible V4 value
   Dom(V3) = {}
5. DWO

Example: N-Queens GAC search Space



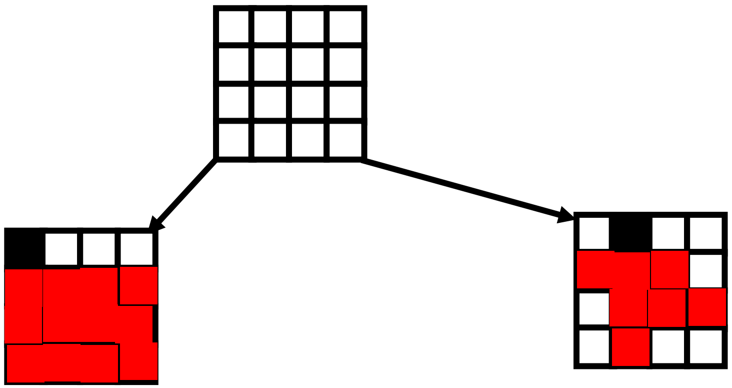
1. V2=1, V2=2, V2=3, V3=2, V3=4, V4=2, all pruned.
   Incompatible with Dom(V1) = {2}
   Dom(V2) = {4}, Dom(V3) = {1,3}, Dom(V4) = {1,3,4}

Example: N-Queens GAC search Space



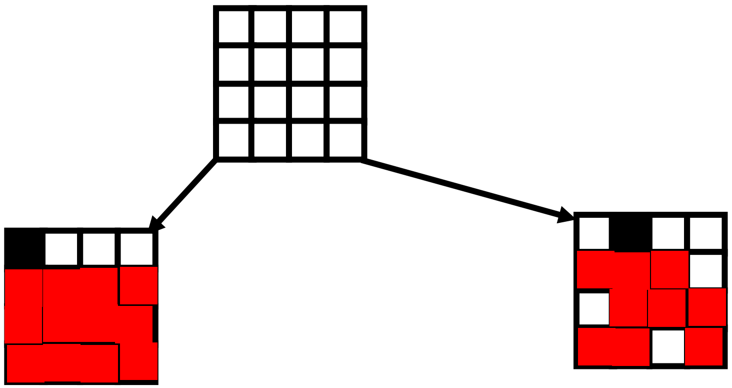
1. V2=1, V2=2, V2=3, V3=2, V3=4, V4=2, all pruned.
   Incompatible with Dom(V1) = {2}
   Dom(V2) = {4}, Dom(V3) = {1,3}, Dom(V4) = {1,3,4}
2. V3=3 has no compatible V2 value
   Dom(V3) = {1}

Example: N-Queens GAC search Space



1. V2=1, V2=2, V2=3, V3=2, V3=4, V4=2, all pruned. Incompatible with Dom(V1) = {2}
   Dom(V2) = {4}, Dom(V3) = {1,3}, Dom(V4) = {1,3,4}
2. V3=3 has no compatible V2 value
   Dom(V3) = {1}
3. V4=1 has no compatible V3 value.
   V4=4 has no compatible V2 value.
   Dom(V4) = 3

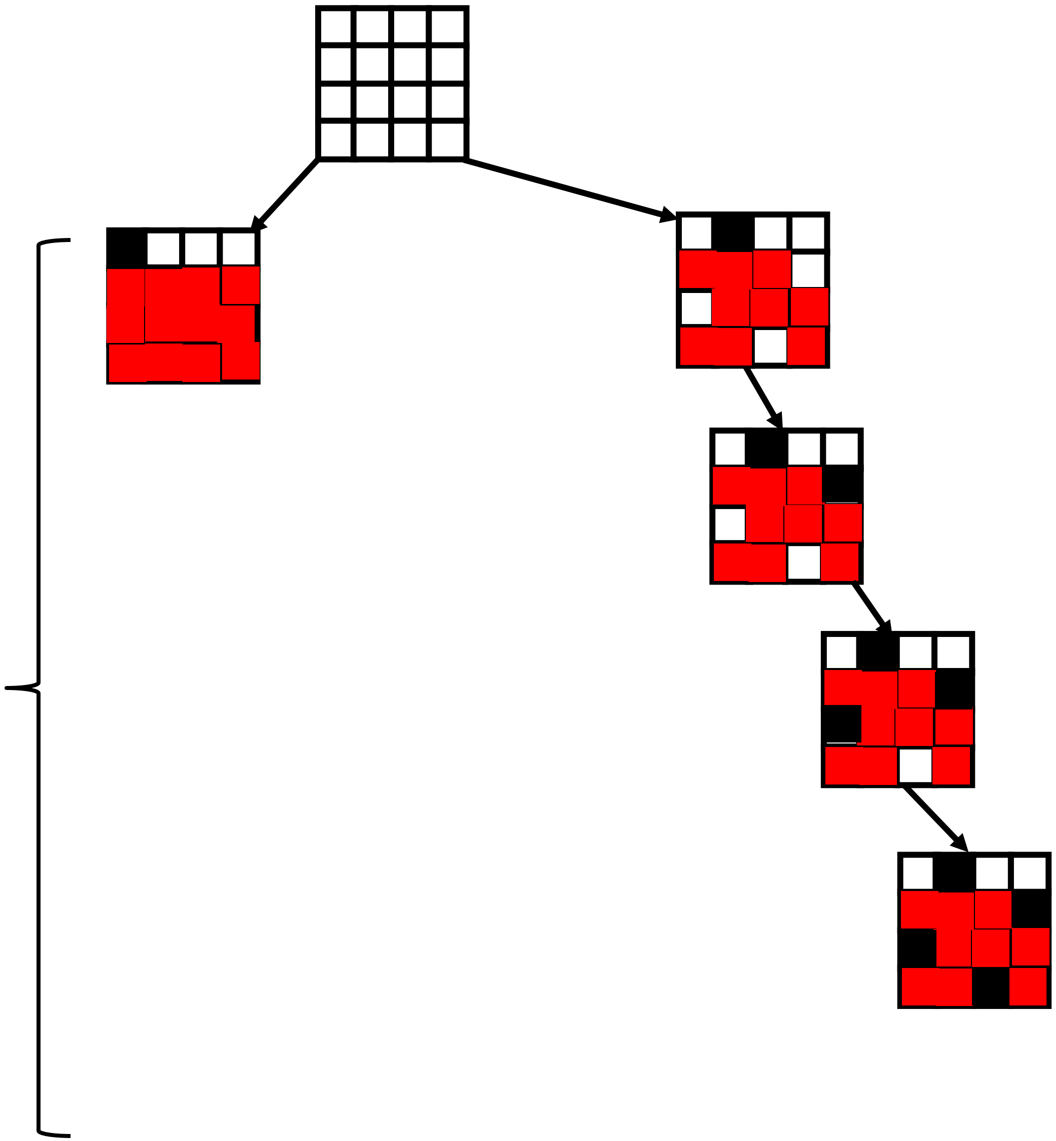
Example: N-Queens GAC search Space



1. V2=1, V2=2, V2=3, V3=2, V3=4, V4=2, all pruned. Incompatible with
   Dom(V1) = {2}
   Dom(V2) = {4}, Dom(V3) = {1,3}, Dom(V4) = {1,3,4}
2. V3=3 has no compatible V2 value
   Dom(V3) = {1}
3. V4=1 has no compatible V3 value.
   V4=4 has no compatible V2 value.
   Dom(V4) = 3

Now search no longer has to branch—only one value left for each variable. So it just walks down to a solution assigning each variable in turn.

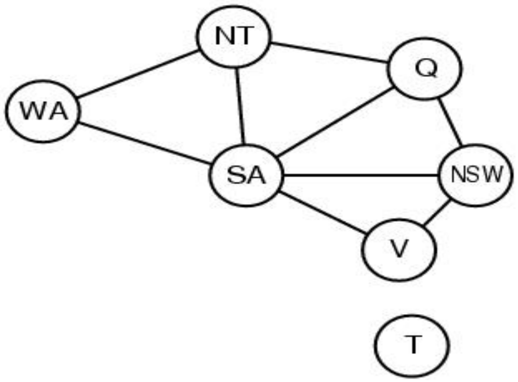
Example: N-Queens GAC search Space



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 SA=B has no compatible NT value

Note Forward Checking would not have detected this DWO.

GAC Algorithm

- Like FCcheck, GacEnforce makes all constraints GAC by pruning variable values.
- GacEnforce does much more pruning than FCCheck.
- As we noted before, making one constraint GAC might make another constraint no longer GAC—so GacEnforce must continually iterate over the constraints until all are GAC.
  - This is accomplished by having a list of constraints that need to be made GAC.
  - Constraints are added back to the list if they might no longer be GAC
  - GacEnforce stops when the list is empty.

GAC Algorithm

- Say a constraint C(V1, V2, …, Vk) is GAC. What can make it non-GAC?
- **GAC condition**: for every value di ∈ curDomain(Vi) there exists values in d1 ∈ curDomain(V1), …, di-1 ∈ curDomain(Vi-1), di+1 ∈ curDomain(Vi+1), …, dk ∈ curDomain(Vk) such that

  C(V1=d1, V2=d2, …, Vi=di, …, Vk=dk) = True

- This tuple of values (one for every variable, and with Vi=di) is said to be a **support** or a **supporting tuple** for Vi=di in constraint C.
- A constraint C is GAC if for every variable Vi in its scope, every value di ∈ curDomain(Vi) has a **support** in C

GAC Algorithm

- So a constraint can only become non-GAC (when previously it was GAC) when some variable in its scope has a value that loses support.
- How can support be lost?
  - The constraint doesn't change—it will still be satisfied by the supporting tuple.
  - But the tuple might have an assignment Vi=di such that di is no longer in curDomain(Vi)
- So if a variable in its scope has a value pruned from its domain ➔ the constraint might no longer be GAC.
- If no values have been pruned ➔ constraint is still GAC

We use this insight to decide when to put a constraint back on the list to be checked.

GAC—Support Functions

- Additional member function for **Constraint class**
  - **.hasSupport(var, val)**
    Returns true if var=val has a supporting tuple in constraint.

Can be implemented in various ways:

1. Iterate over all possible assignments to the other variables in the constraint, (values from their current domain) to see if any combination along with var=val satisfies the constraint.
2. Store all the supporting tuples for var=val (and for each variable/value pair). Check if any of these are still in the variable's current domains.
3. Store one supporting tuple for var=val, if that one has been lost, use a method like (1) to find a new one.
4. **A range of clever implementation methods have been developed**

Complexity grows exponentially with constraint's arity. (We have to check all possible combination of assignments to the other variables).

Enforce GAC (prune all GAC inconsistent values)

```
GacEnforce(cnstrs,assignedvar,assignedval):
  #cnstrs is a collection of constraints not known GAC.
  #Establish GAC on them and on all affected constraints
  while not cnstrs.empty():
    cnstr = cnstrs.extract()      #make cnstr GAC
    for var in cnstr.scope():
      for val in var.curDomain():
        if not cnstr.hasSupport(var,val):
          var.pruneValue(val,assignedvar,assignedval)
          if var.curDomainSize() == 0:
            return ″DWO″  #domain wipe out
          for recheck in constraintsOf(var):
            if recheck != cnstr and not recheck in cnstrs:
              cnstrs.insert(recheck)
  return ″OK″
```

GAC Algorithm, enforce GAC during search

```
GAC(unAssignedVars):  #search while maintaining GAC
  if unAssignedVars.empty():
    for var in variables:
      print var.name(), " = ", var.getValue()
    if allSolutions:
      return     #continue search to print all solutions
    else: EXIT   #terminate after one solution found
  var := unAssignedVars.extract()   #select next variable to assign
  for val in var.curDomain():       #current domain!
    var.setValue(val)
    noDWO = True
    if GacEnforce(constraintsOf(var),var, val) == "DWO":
      #only var's domain changed—constraints with var have to be checked
      noDWO = False
    if noDWO:
        GAC(unAssignedVars)
    restoreValues(var, val)        #restore values pruned by var = val assignment
  var.setValue(None)               #set var to be unassigned and return to list
  unAssignedVars.insert(var)
  return
```

GAC–Initial call

- Initially all variables are unassigned
- Initially unAssignedVars contains all variables
- Initially all constraints are GAC: this is accomplished by the call

  GacEnforce(**constraints**, None, None)

- Remember the variable **constraints** is a list containing all constraints of the CSP.
- By passing the pruning “reason” as the ”None” assignment, the values pruned at the root will not be in the variable’s current domains during search.

.hasSupport

- Finding a support for V=d in constraint C in the worst case requires $O(2^k)$ work, where k is the arity of C, i.e., |scope(C)|.
- So the method of examining alternate assignments to the other variables is limited to constraints of relatively small arity.

- **A key development** in practice is that for some constraints this computation can be done time polynomial in the constraint's arity.
- An important example is the all-diff constraint. For, 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 matching theory.
- The special purpose algorithms for achieving GAC on particular types of constraints are very important in practice.

GAC enforce example

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

= 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_{SS8}) $\rightarrow$ 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}) $\rightarrow$ CurDom of $V_{1,7}$, $V_{1,8}$ = {2, 3, 7, 9}
CurDom of $V_{1,5}$, $V_{1,6}$, = {2, 3}

GAC(C_{SS8}) $\rightarrow$ CurDom of $V_{2,4}$, $V_{3,4}$, $V_{3,6}$ = {1, 4, 8}

GAC(C_{C5}) $\rightarrow$ CurDom of $V_{5,5}$,$V_{9,5}$ = {2, 6, 8}
$\rightarrow$ CurDom of $V_{1,5}$ = {2}

GAC(C_{SS8}) $\rightarrow$ 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