Constraint satisfaction problem (CSP)

- Formalization of a CSP
- Backtracking algorithm
- Forward checking algorithm
- GAC algorithm

Next: Knowledge representation and reasoning (KR)

Readings: Chapter 8, 9

*Slides based on those of Sheila McIlraith

Constraint Satisfaction Problems

- The search algorithms we discussed so far had no knowledge of the states representation (black box).
- Instead we can have a general state representation that works well for many different problems.
- We can build then specialized search algorithms that operate efficiently on this general state representation.
- We call the class of problems that can be represented with this specialized representation CSPs – Constraint Satisfaction Problems.

CSPs: State Representation

The idea: represent states as a vectors of feature values

- A set of k features (or variables)
- Each variable has a domain of different values, *e.g.*
  - height $=$ {short, average, tall},
  - weight $=$ {light, average, heavy}
- A state is specified by an assignment of a value for each variable.
- A partial state is specified by an assignment of a value to some of the variables.
- a goal state specified as conditions on the vector of feature 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: For cells that are already filled in, the domain is the singleton set containing the predefined value. The domain of other variables is the set $\{1\text{-}9\}$.
- 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.
- Solution: a value for each cell satisfying the constraints:
  - no cell in the same column can have the same value.
  - no cell in the same row can have the same value.
  - no cell in the same sub-square can have the same value.

Example: Scheduling

Want to schedule a time and a space for each final exam so that

- No student is scheduled to take more than one final at the same time.
- The space allocated has to be available at the time set.
- The space has to be large enough to accommodate all of the students taking the exam.

Example: Scheduling

We use the index i to uniquely identify exams – the "i-th exam".

- T_i is a variable representing the scheduled time for the i-th final exam.
  - Domains of T_i are fixed to {FriAm, TuesPm, MonAm, MonPm, $\cdots$, FriPm}.
- S_i is the space variable for the i-th final.
  - Domain of S_i are all rooms big enough to hold the i-th final
- For all pairs of finals i, j $(i \neq j)$ such that there is a student taking both: $T_i \neq T_j$
- For all pairs of finals i, j $(i \neq j)$: $T_i \neq T_j$ or $S_i \neq S_j$

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
  - $V_i = d$ where $d \in Dom[V_i]$
- Each constraint C
  - has a set of variables it is over, called its scope, *e.g.*, $C(V_1, V_2, V_4)$
  - Acts as a boolean function that maps assignments to these variables to true/false, *e.g.*,
    - $C(V_1 = a, V_2 = b, V_4 = c) = True$
    - $C(V_1 = b, V_2 = c, V_4 = c) = False$

Constraints

- 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 (n-ary) constraints: over 3 or more variables.

Specification of constraints

We can specify the constraint with a table

E.g., 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

Specification of constraints

Often we can specify the constraint more compactly with an expression:

E.g., 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**:
  - $Dom[V_{ij}] = \{1\text{-}9\}$ 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($V_{14}, V_{15}, V_{16}, \ldots, V_{34}, V_{35}, V_{36}$)

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

Solving CSPs

- 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.
- Thus CSPs can be solved by a specialized version of depth first search.

Solving CSPs

- We can build up to a solution by searching through the space of partial assignments.
- In principle, the order in which we assign the variables does not matter – eventually they all have to be assigned.
- If we falsify a constraint during the process of building up a solution, we can immediately reject the current partial assignment.

CSP as a 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

A generic backtracking algorithm

- We pick a variable*,
- pick a value for it*,
- test the constraints that we can,
- if a constraint is unsatisfied we backtrack,
- otherwise we set another variable.
- When all the variables are set, we're done.

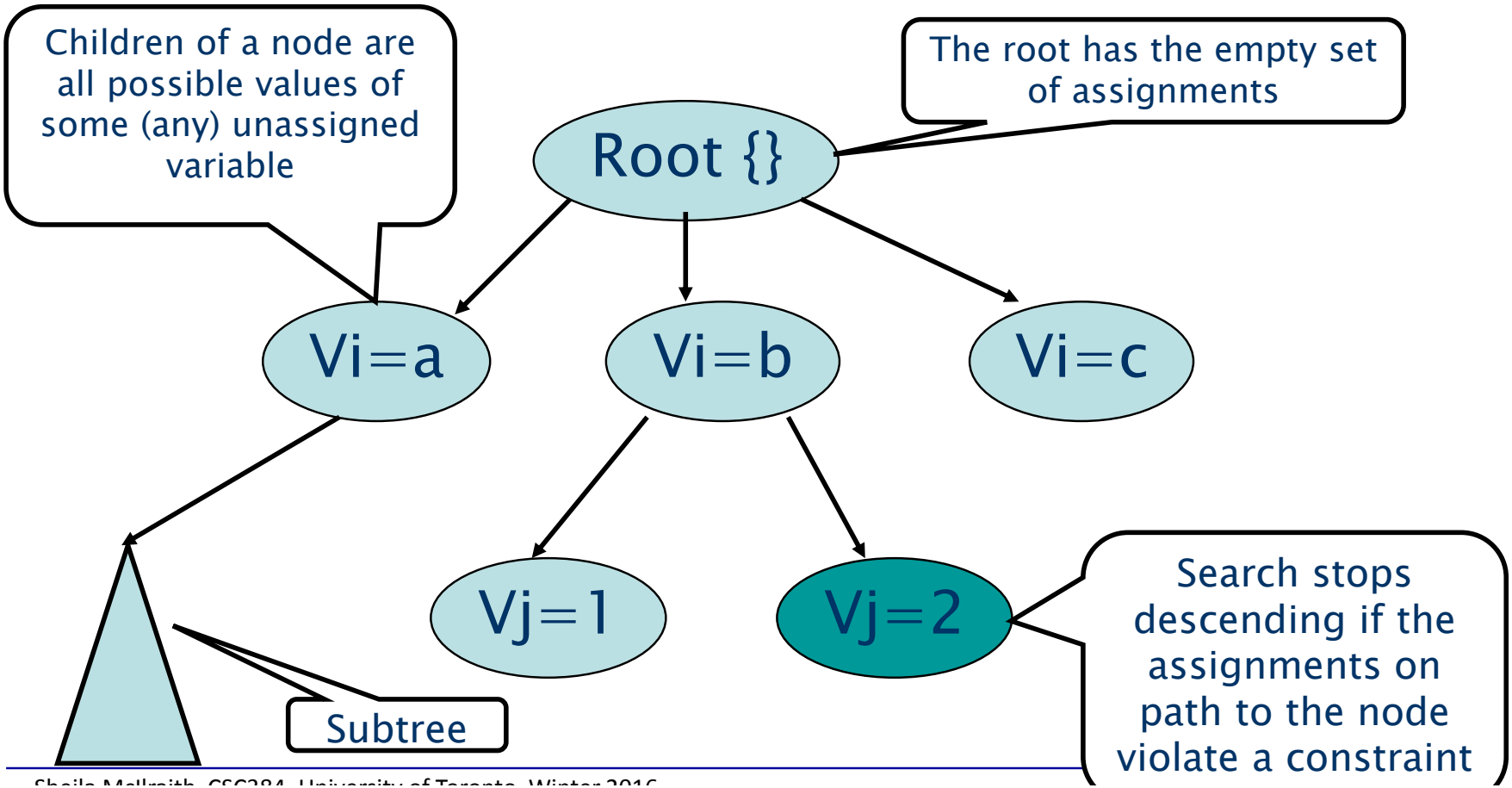
Backtracking Search: The Algorithm BT

```
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:
                 ;(rarely the case initially high in the search tree)
            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.

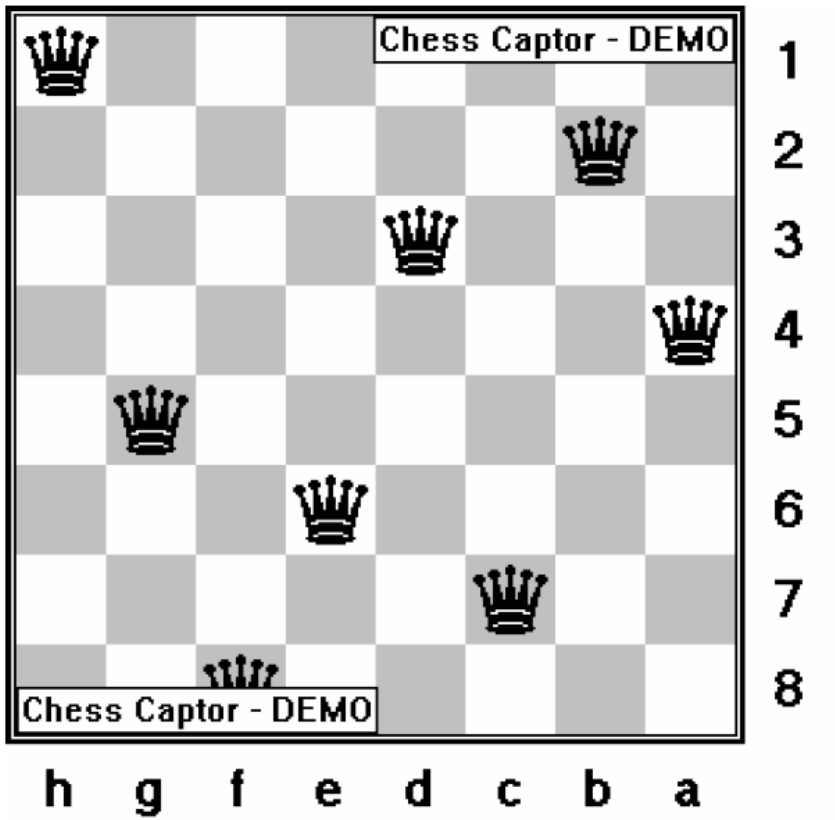


Backtracking Search

- 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.

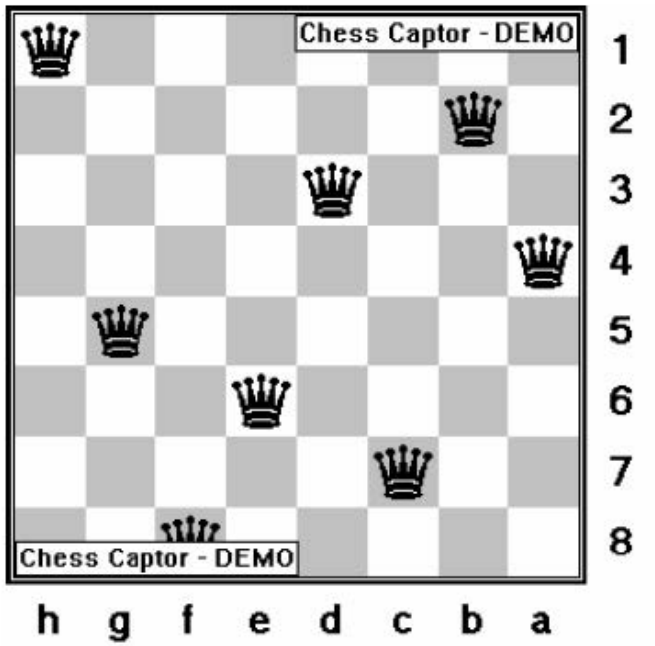
Example: N-Queens

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



Formulation 1

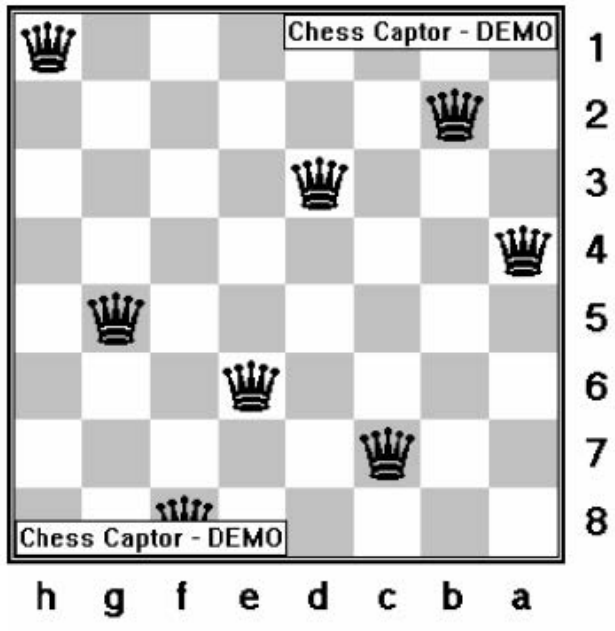
- N variables, N^2 values for each variable representing the positions on the chessboard
- *e.g.*, Q1 = 1, Q2 = 15, Q3 = 21, Q4 = 32, Q5 = 34, Q6 = 44, Q7 = 54, Q8 = 59



- This representation has N^{2N} states
  - 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

Formulation 2

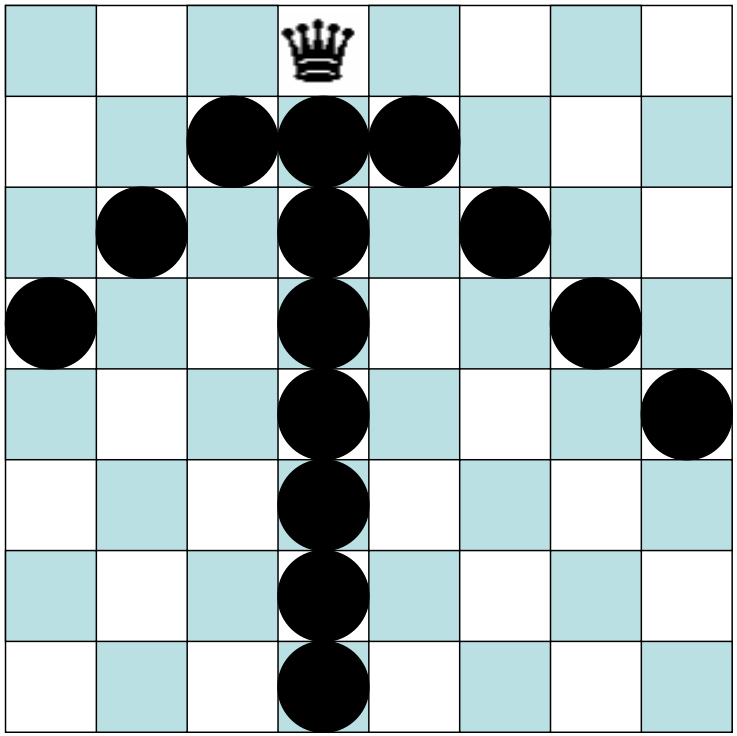
- N variables Q_i, value of Q_i is the column the Queen in row i is placed; possible values $\{1, \cdots, N\}$.
- *e.g.*, Q1 = 1, Q2 = 7, Q3 = 5, Q4 = 8, Q5 = 2, Q6 = 4, Q7 = 6, Q8 = 3



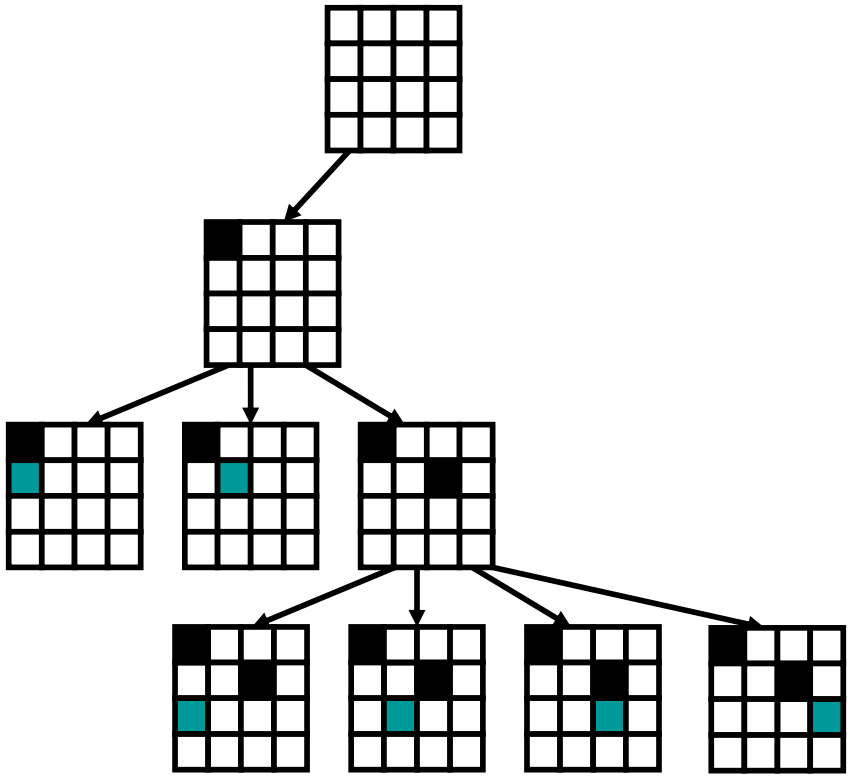
- This representation has N^N states
  - For 8-Queens: 8^8 = 16,777,216
- The choice of a representation can make the problem solvable or unsolvable!

Constraints

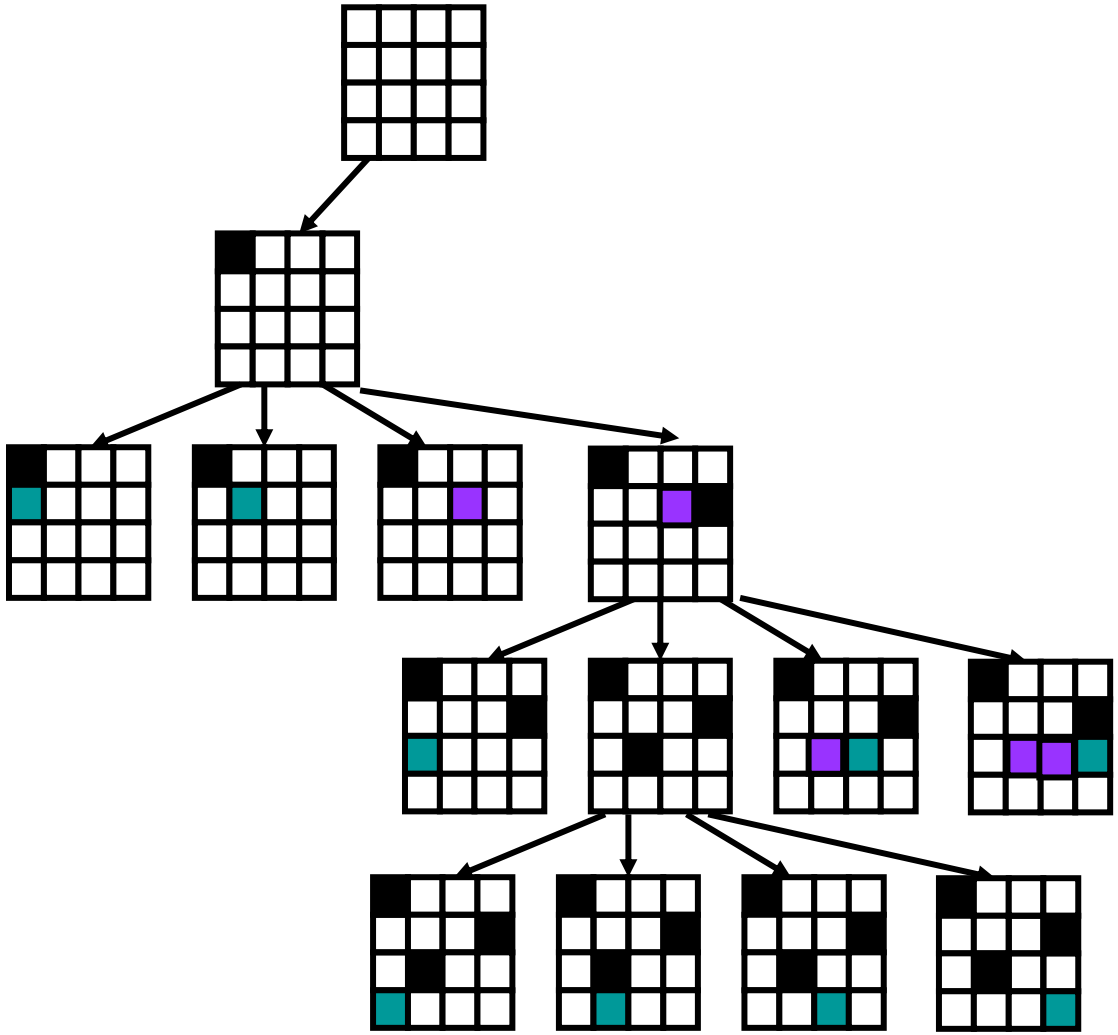
- Can't put two Queens in same column: $Q_i \neq Q_j$ for all $i \neq j$
- Diagonal constraints: $abs(Q_i - Q_j) \neq abs(i - j)$



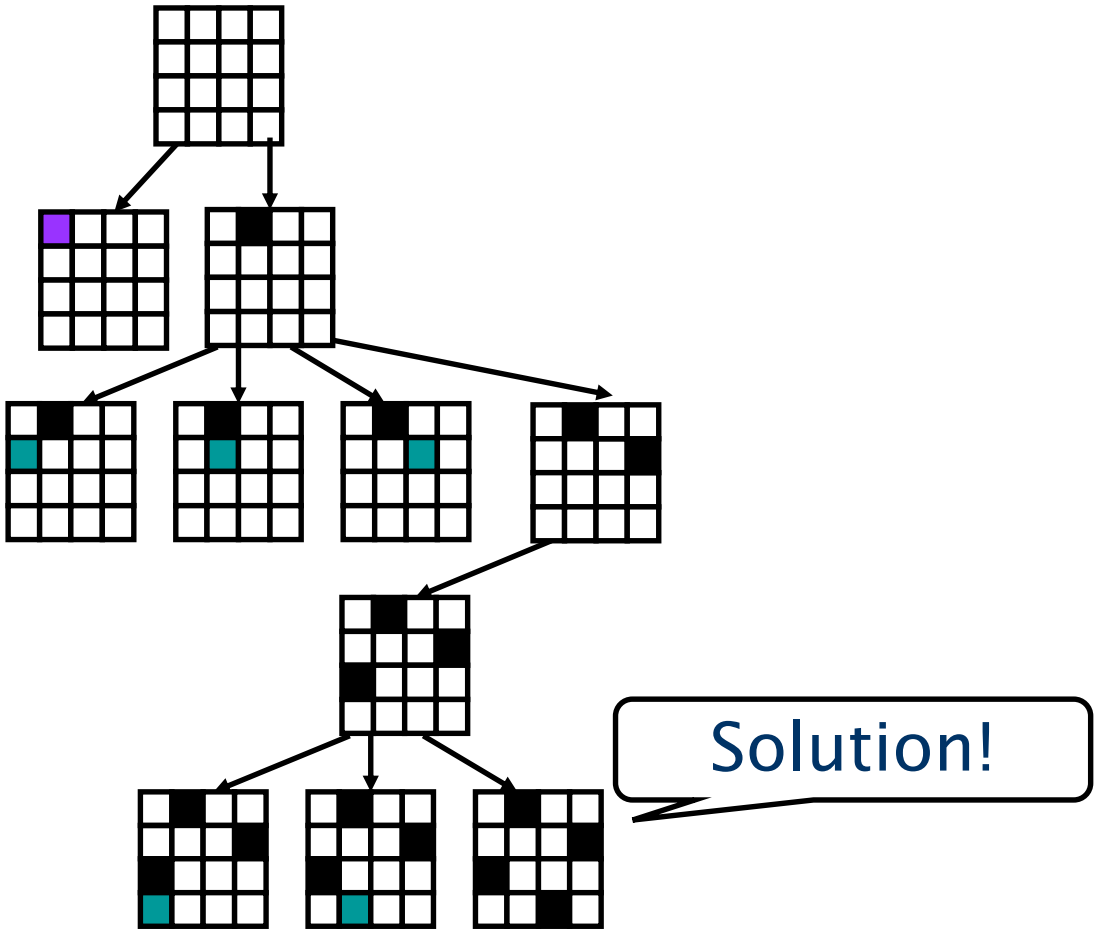
Solving 4-queens



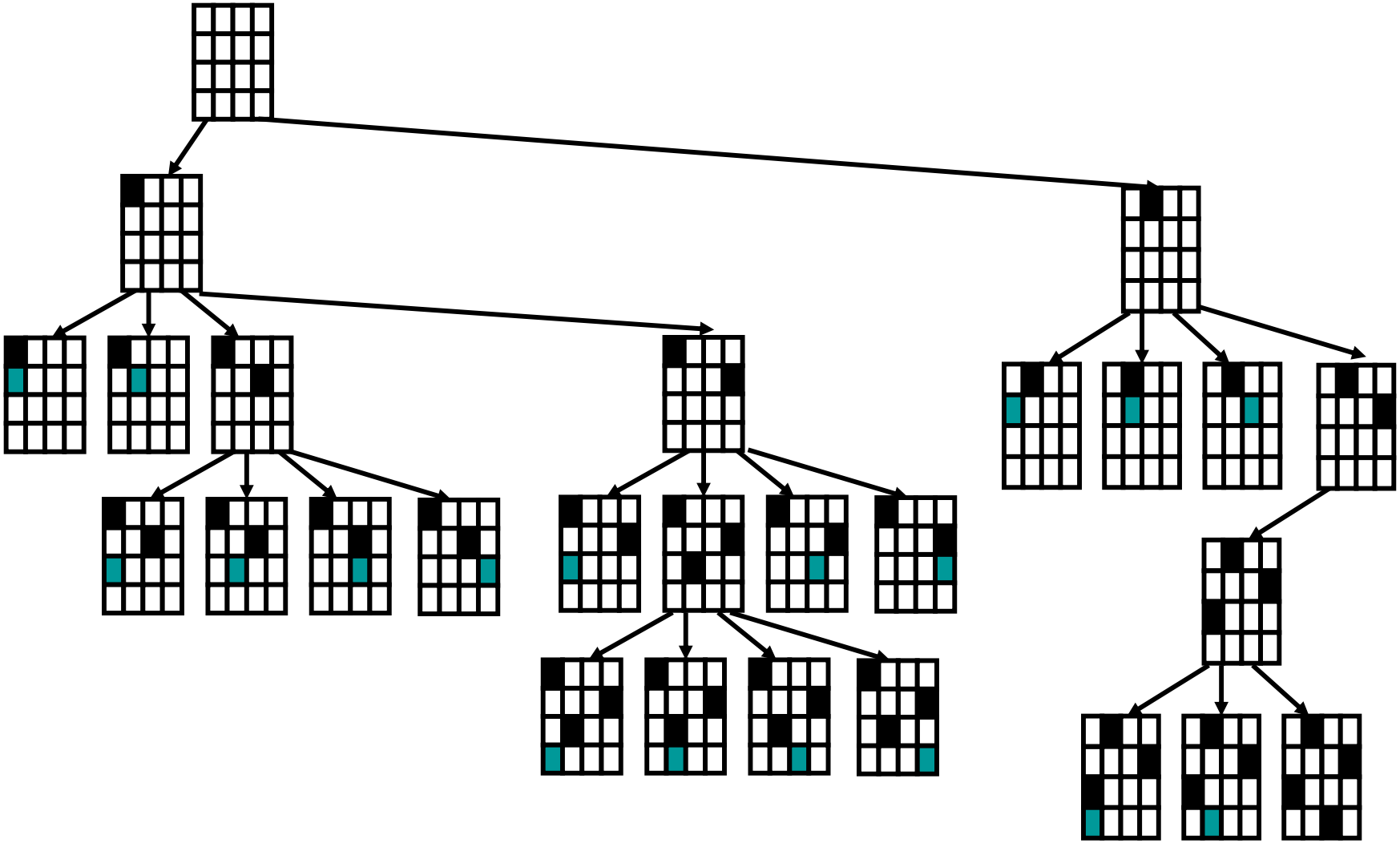
Solving 4-queens



Solving 4-queens



Example: 4-queens backtracking search space



Formalization of a CSP

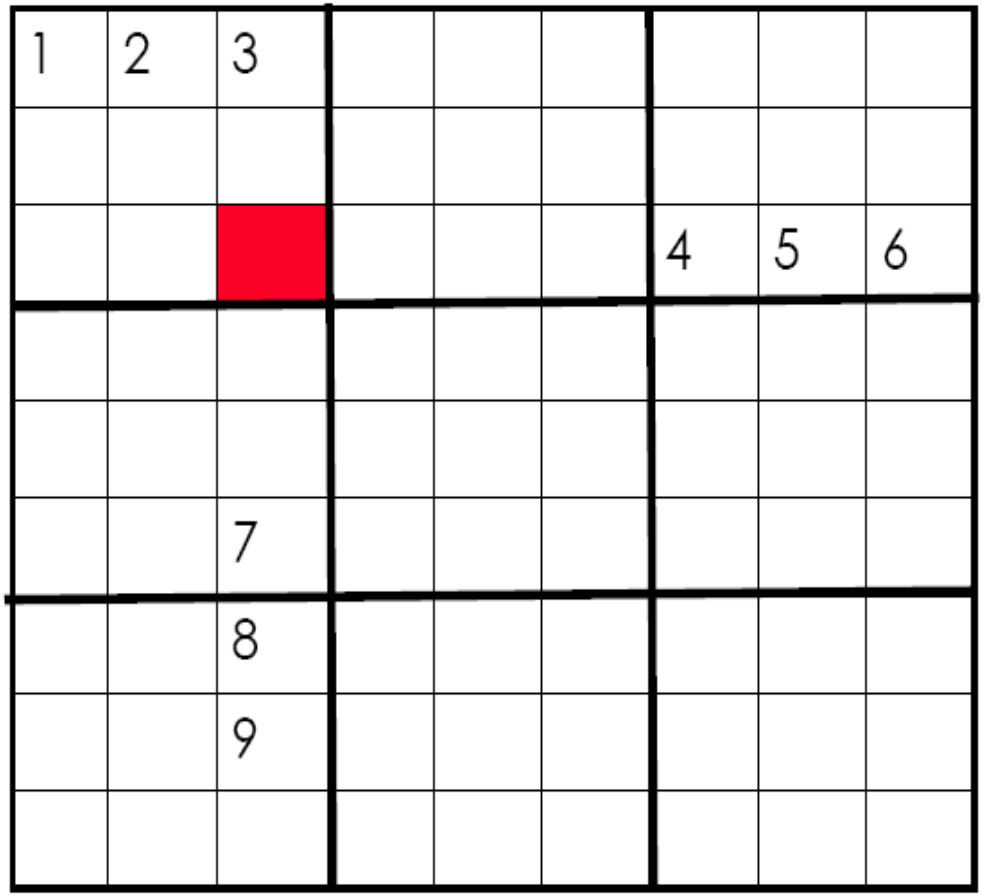
- A CSP consists of
  - A set of variables $V_1, \ldots, V_n$
  - For each variable a (finite) domain of possible values Dom$[V_i]$.
  - A set of constraints $C_1, \ldots, C_m$.
- A solution to a CSP is an assignment of a value to all of the variables such that every constraint is satisfied.
- A CSP is unsatisfiable if no solution exists.

A generic backtracking algorithm

- We pick a variable*,
- pick a value for it*,
- test the constraints that we can,
- if a constraint is unsatisfied we backtrack,
- otherwise we set another variable.
- When all the variables are set, we're done.

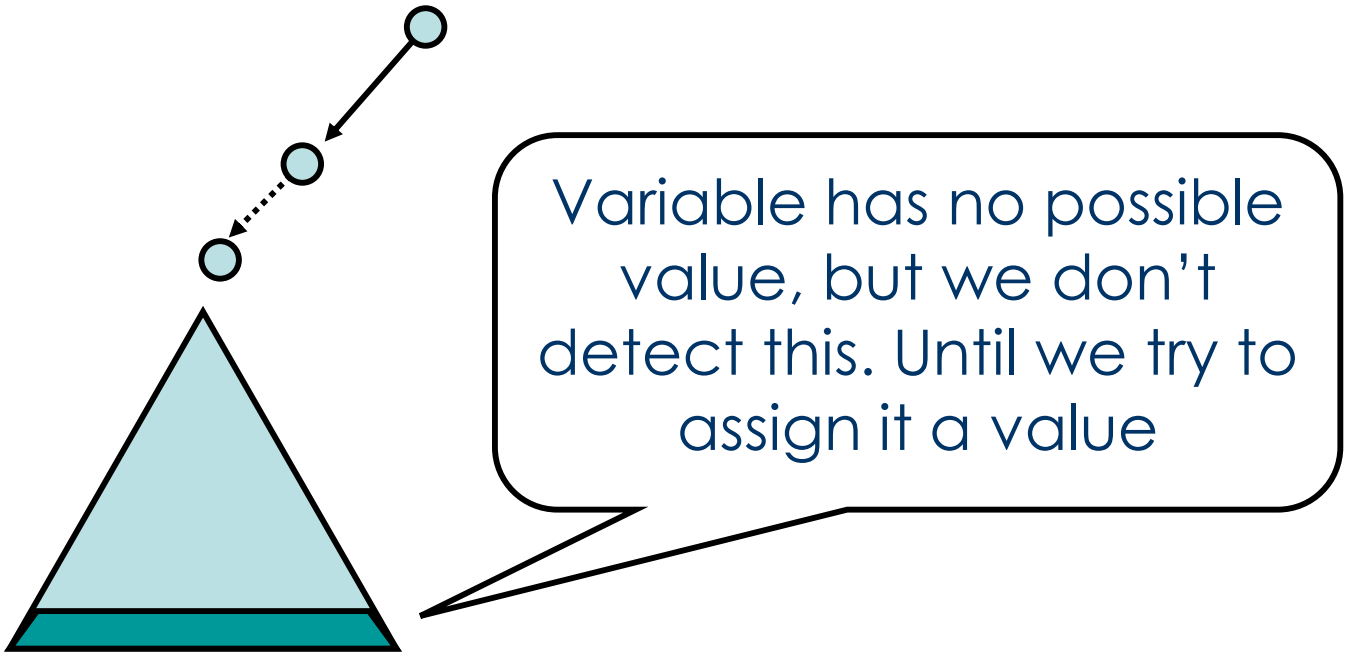
Problems with Plain Backtracking

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



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

Constraint Propagation (Inference)

- 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.
- It can also be applied before search even begins!
- 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:
  1. Forward Checking
  2. Generalized Arc Consistency

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
```

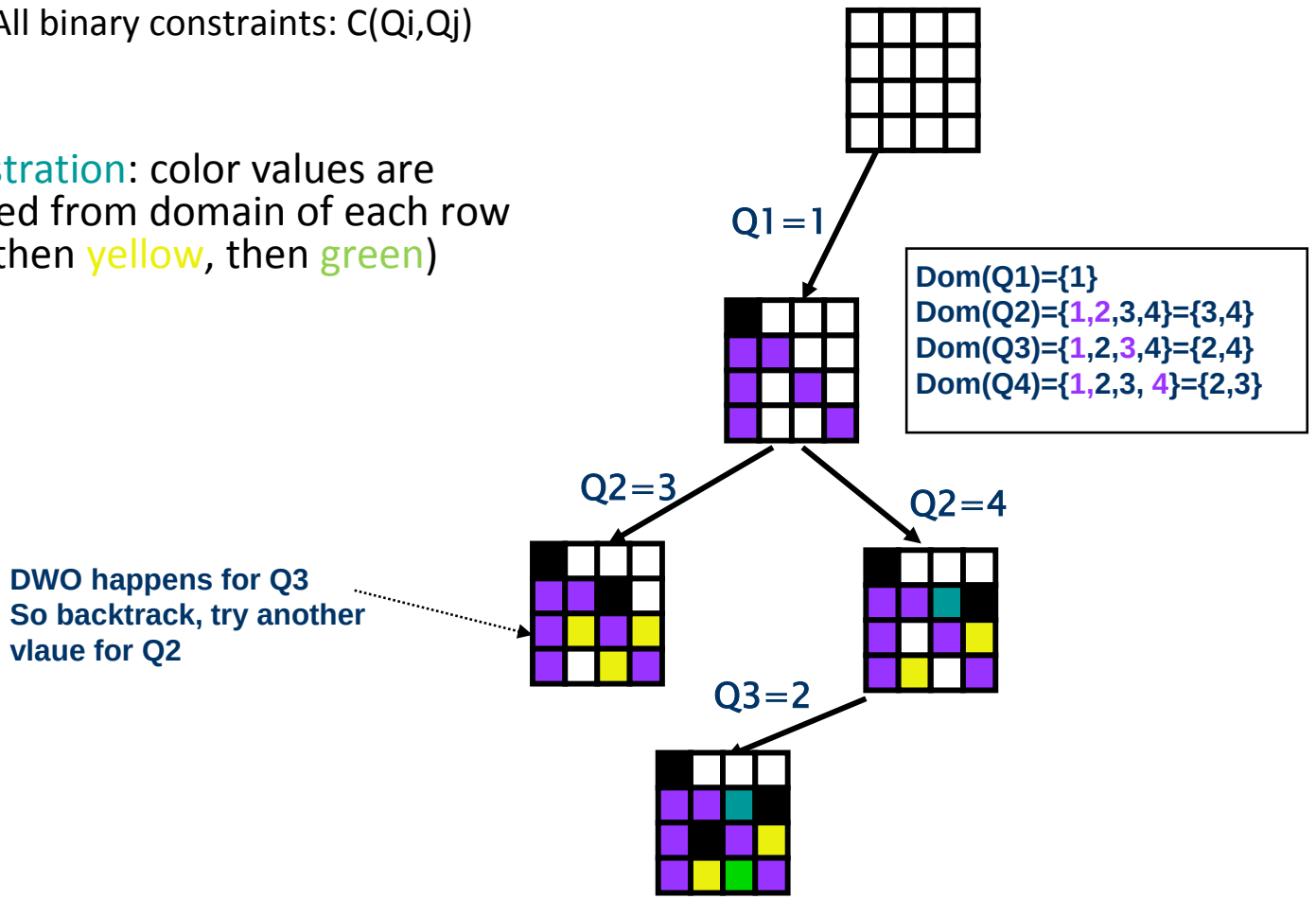
Forward Checking Algorithm

```
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;
```

FC 4-queens

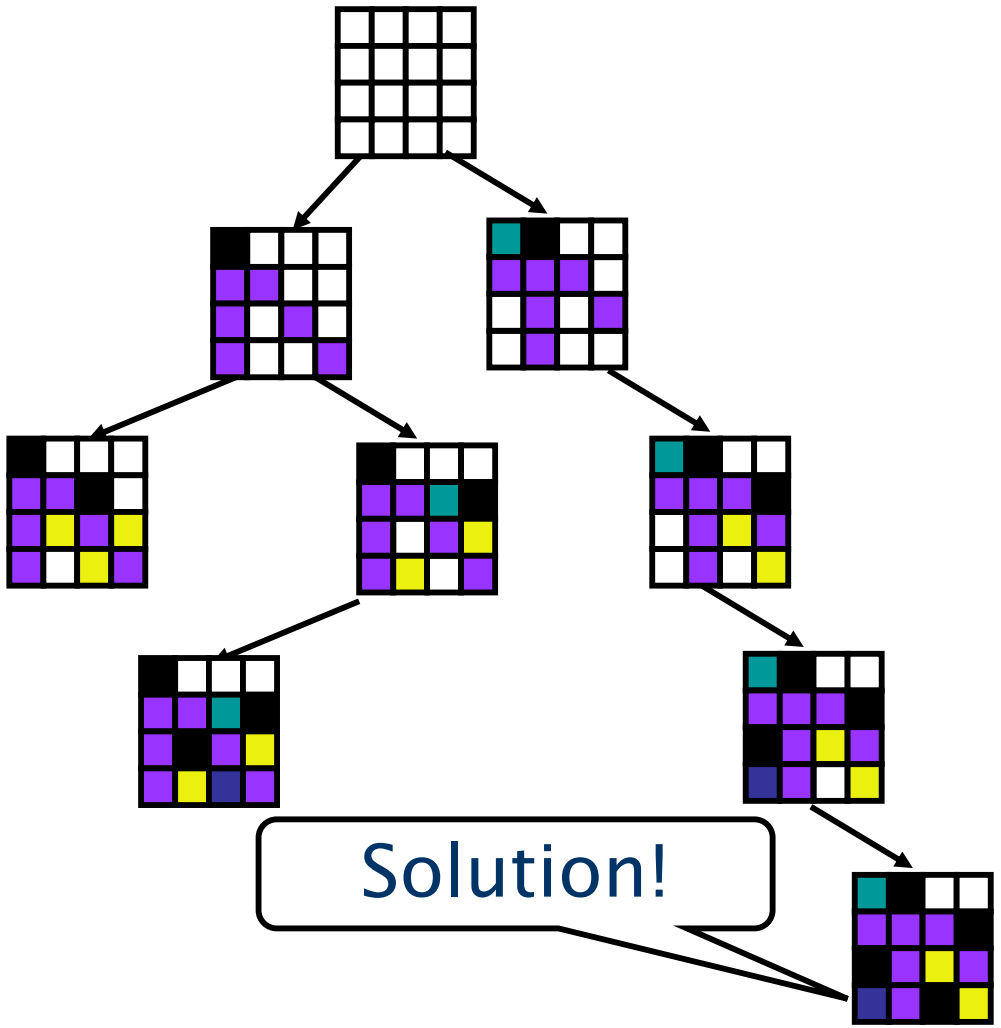
- 4X4 Queens
  - Q1,Q2,Q3,Q4 with domain {1..4}
  - All binary constraints: C(Qi,Qj)

- FC illustration: color values are removed from domain of each row (blue, then yellow, then green)

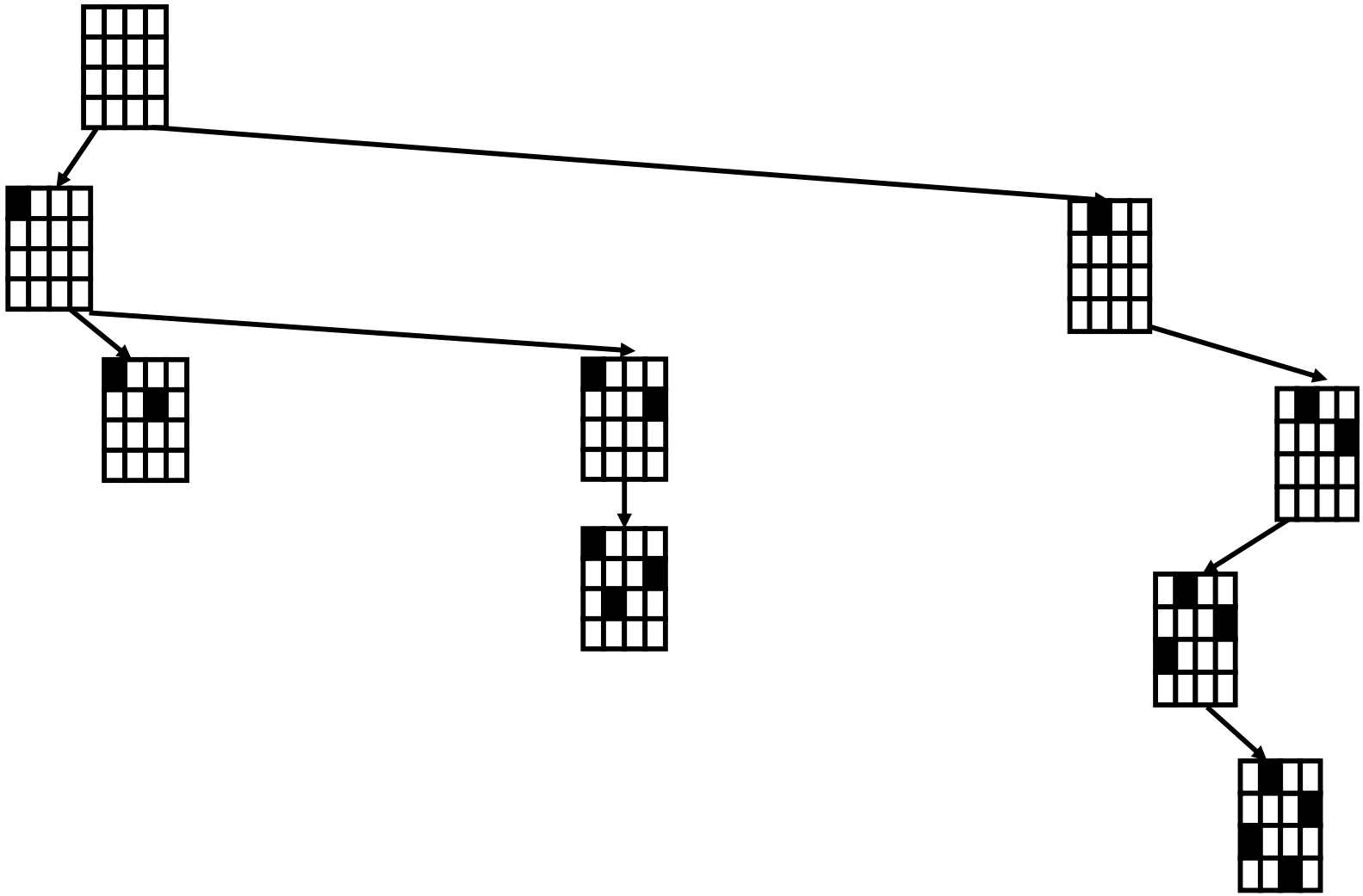


FC 4-queens

- 4X4 Queens continue...



FC 4-queens 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).

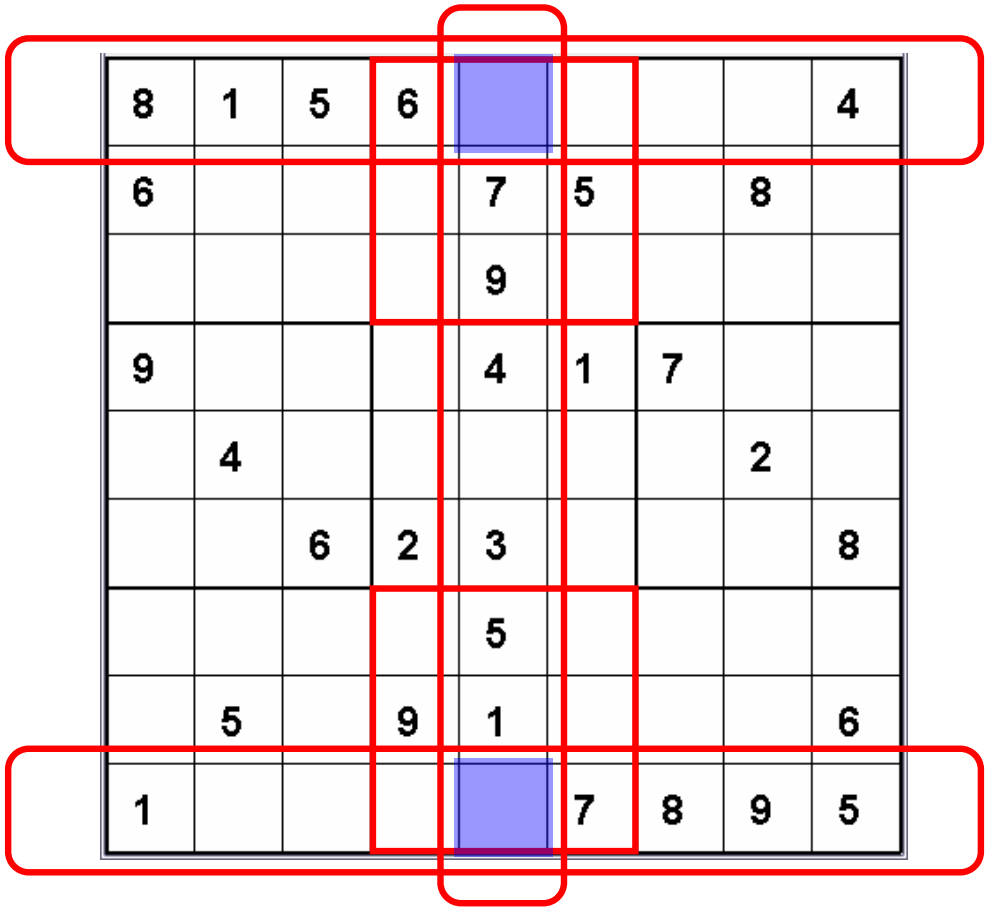
FC: Minimum Remaining Values Heuristics (MRV)

FC also gives us for free a very powerful heuristics regarding 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 can find an 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

Most restricted variables! = MRV

Domain of each variable: {1, …, 9}

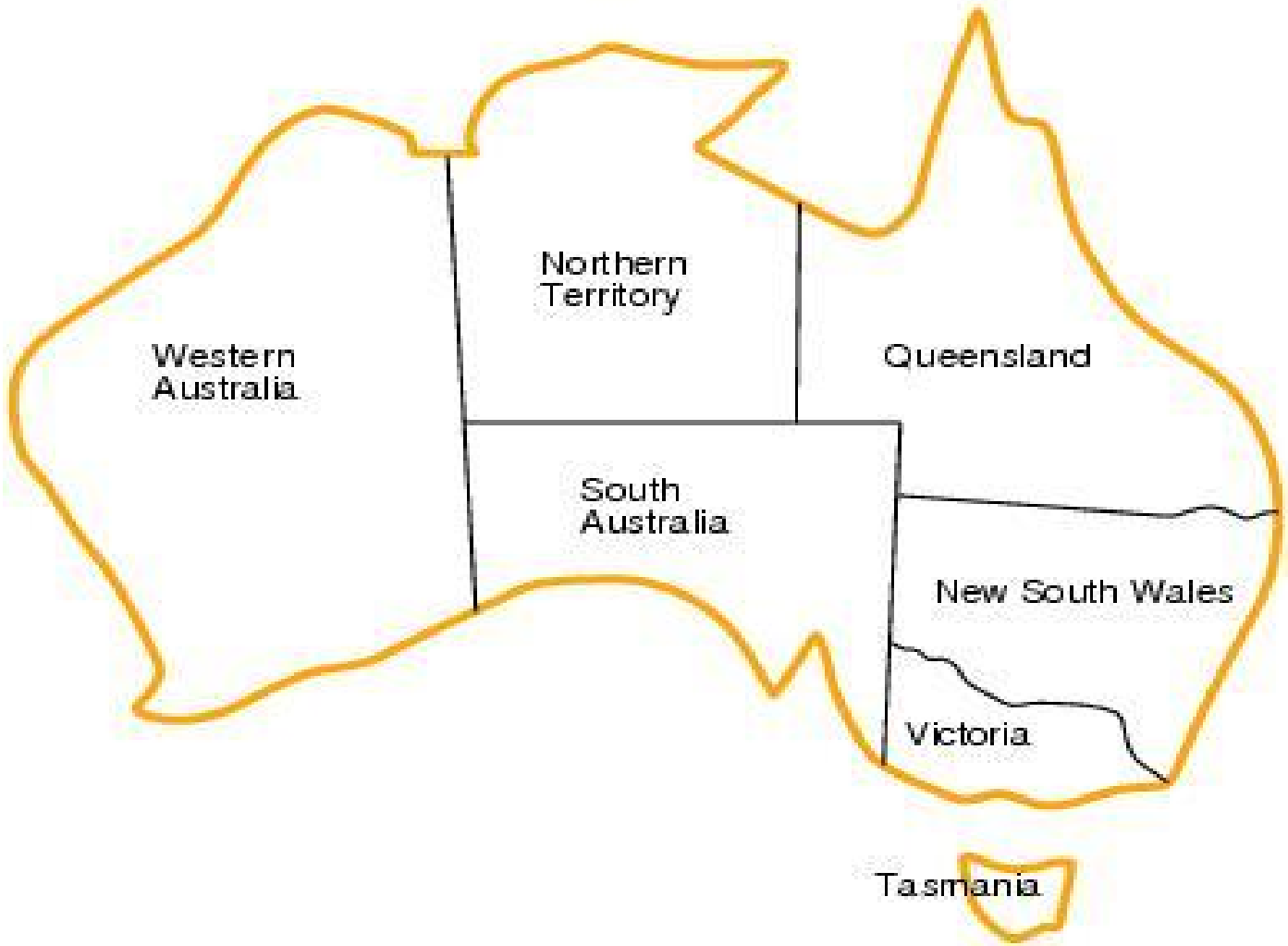
(1, 5) **impossible values:**
Row: {1, 4, 5, 6, 8}
Column: {1, 3, 4, 5, 7, 9}
Subsquare: {5, 6, 7, 9}
→ Domain = {2}

(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 = {6}

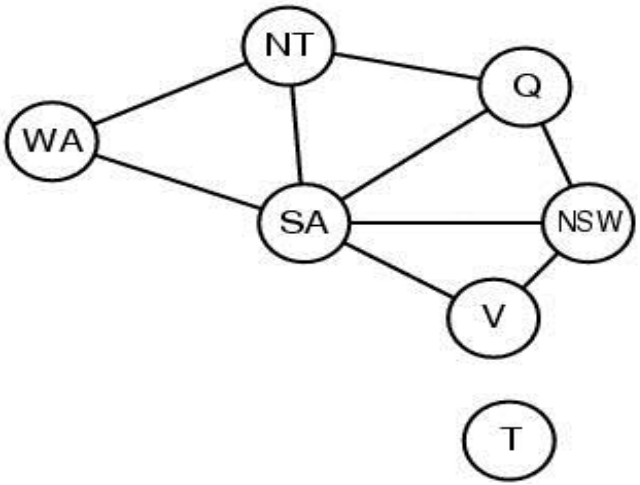
Example – Map coloring

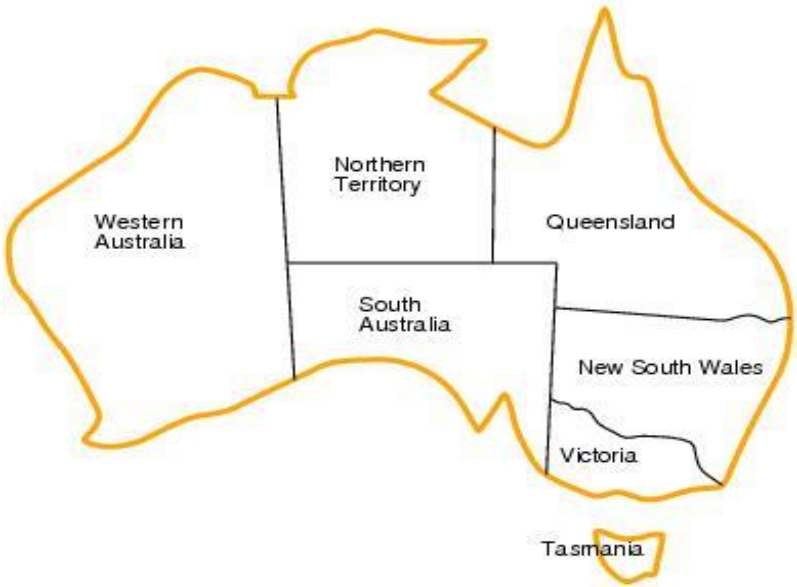
Color the following map using red, green, and blue such that adjacent regions have different colors



Formulation

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

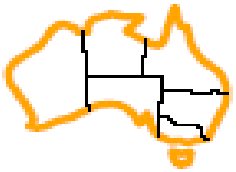


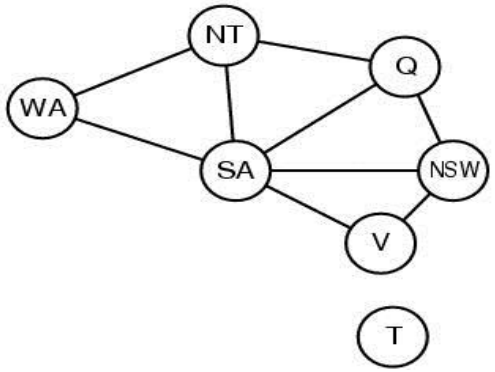


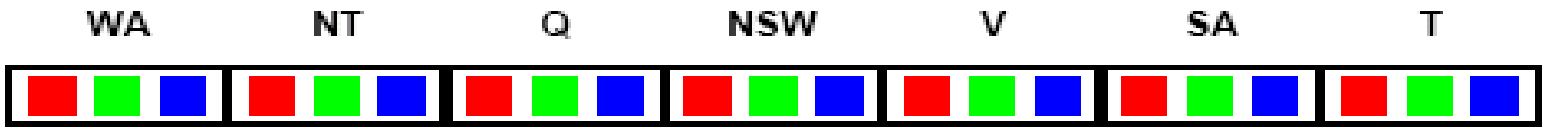
Example – Map coloring

Forward checking idea:

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

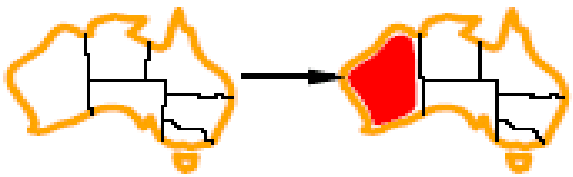


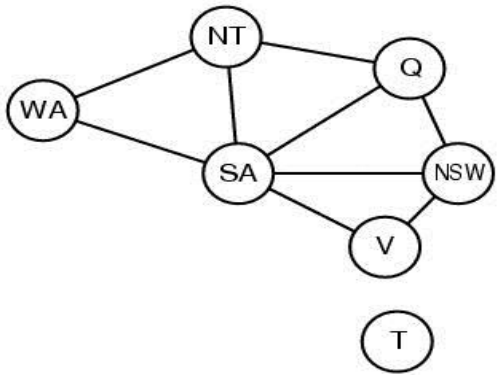


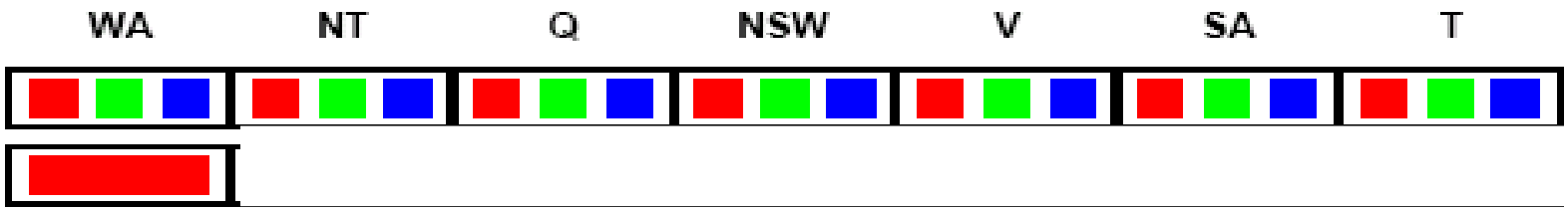


Example – Map coloring

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

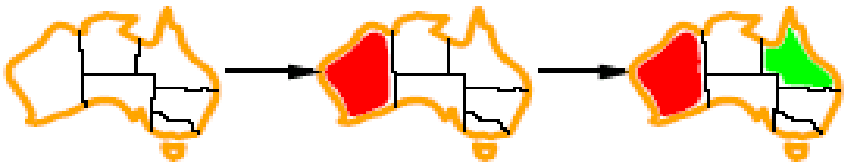


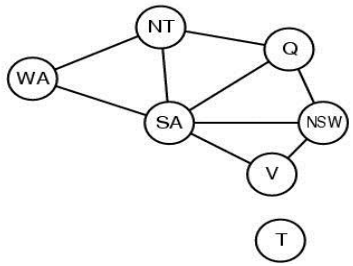


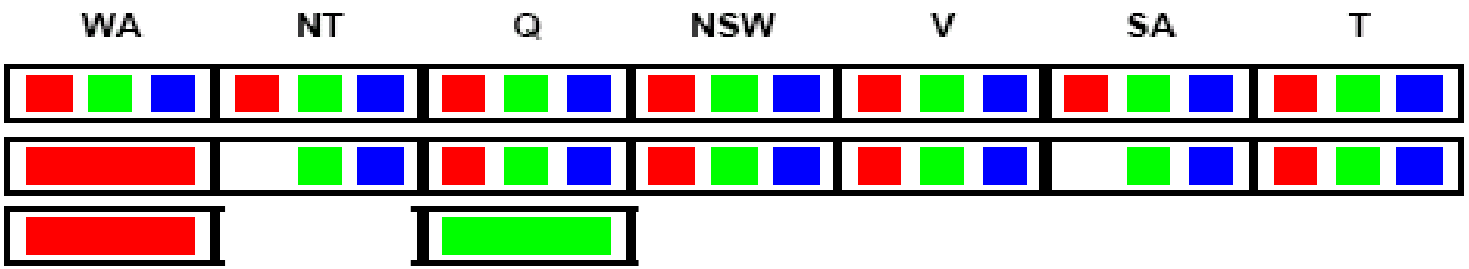


Example – Map coloring

- 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

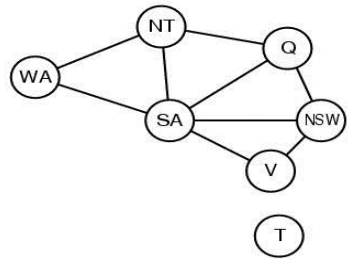






Example – Map coloring

- Assign *{V=blue} (not using MRV)*
- Effects on other variables connected by constraints with V
  - *NSW can no longer be blue*
  - *SA is empty (DWO – Domain Wipe Out!)*
- FC has detected that partial assignment is *inconsistent* with the constraints and backtracking can occur.



Exercise

Try the previous map coloring example with 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.

Generalized Arc Consistency (GAC)

- C(X,Y) is consistent iff for every value of X there is some value of Y that satisfies C
- $C(V_1, V_2, V_3, \ldots, V_n)$ is GAC wrt V_i iff for every value of V_i, there are values of $V_1, \ldots, V_{i-1}, V_{i+1}, \ldots, V_n$ that satisfy C
- A constraint is GAC iff it is GAC wrt every variable in its scope
- A CSP is GAC iff all of its constraints are GAC

GAC

- Say we find a value d of variable V_i that is not consistent wrt a constraint
- *i.e.*, 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 V_i as this value cannot lead to a solution
- Much like Forward Checking, but more powerful
  - If d is removed from $CurDom[V_i]$, it must be the case that d is arc inconsistent

GAC: an example

C(X,Y): $X > Y$, Dom(X)=$\{1,5,11\}$, Dom(Y)=$\{3,8,15\}$

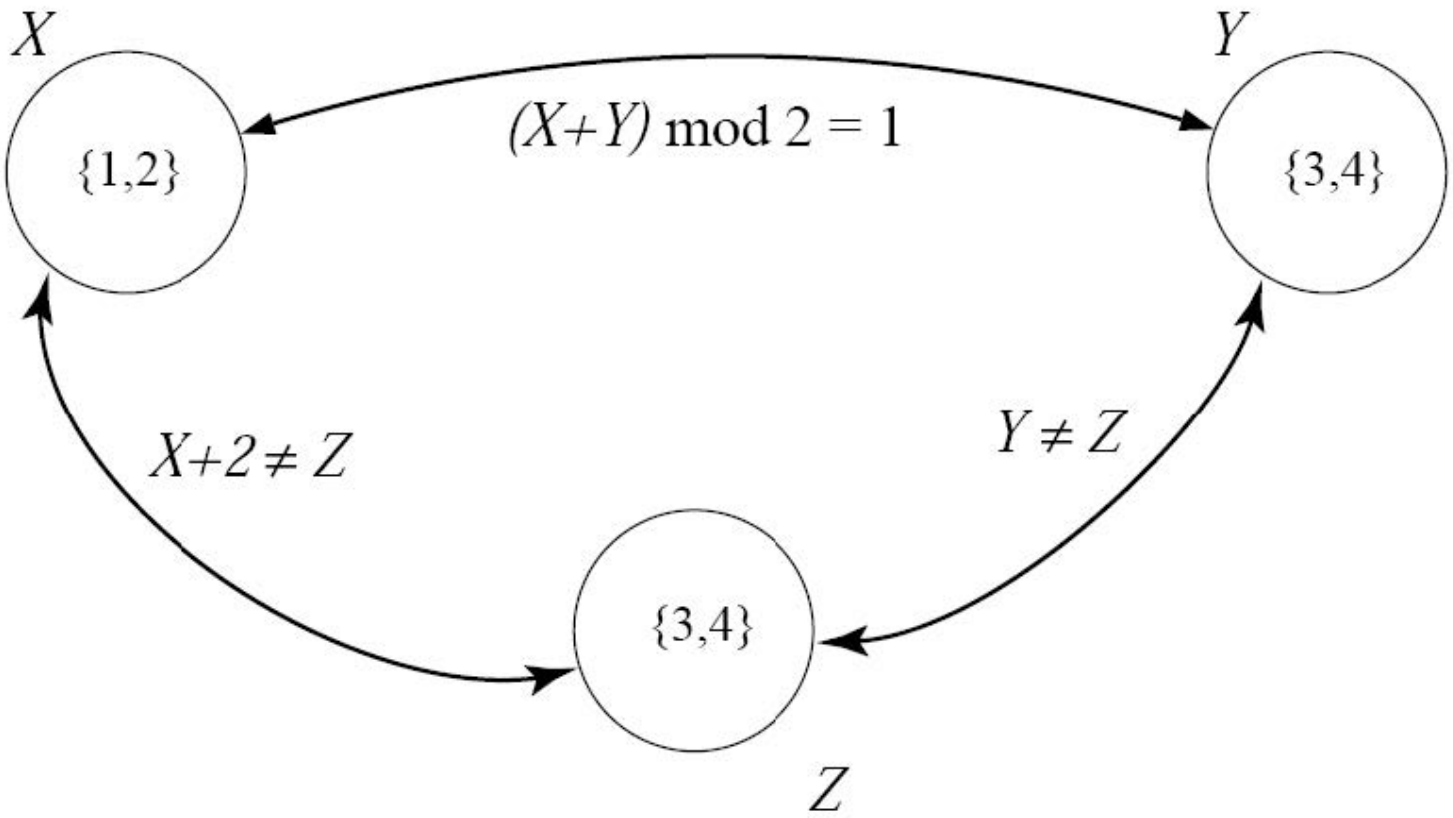
- For X=1 there is no value of Y s.t. $1 > Y$, so remove 1 from domain X
- For Y=15 there is no value of X s.t. $X > 15$, so remove 15 from domain Y
- We obtain Dom(X)=$\{5,11\}$ and Dom(Y)=$\{3,8\}$.

GAC

- Removing a value from a 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
- If we apply arc consistency propagation during search the search tree will typically be much reduced in size.

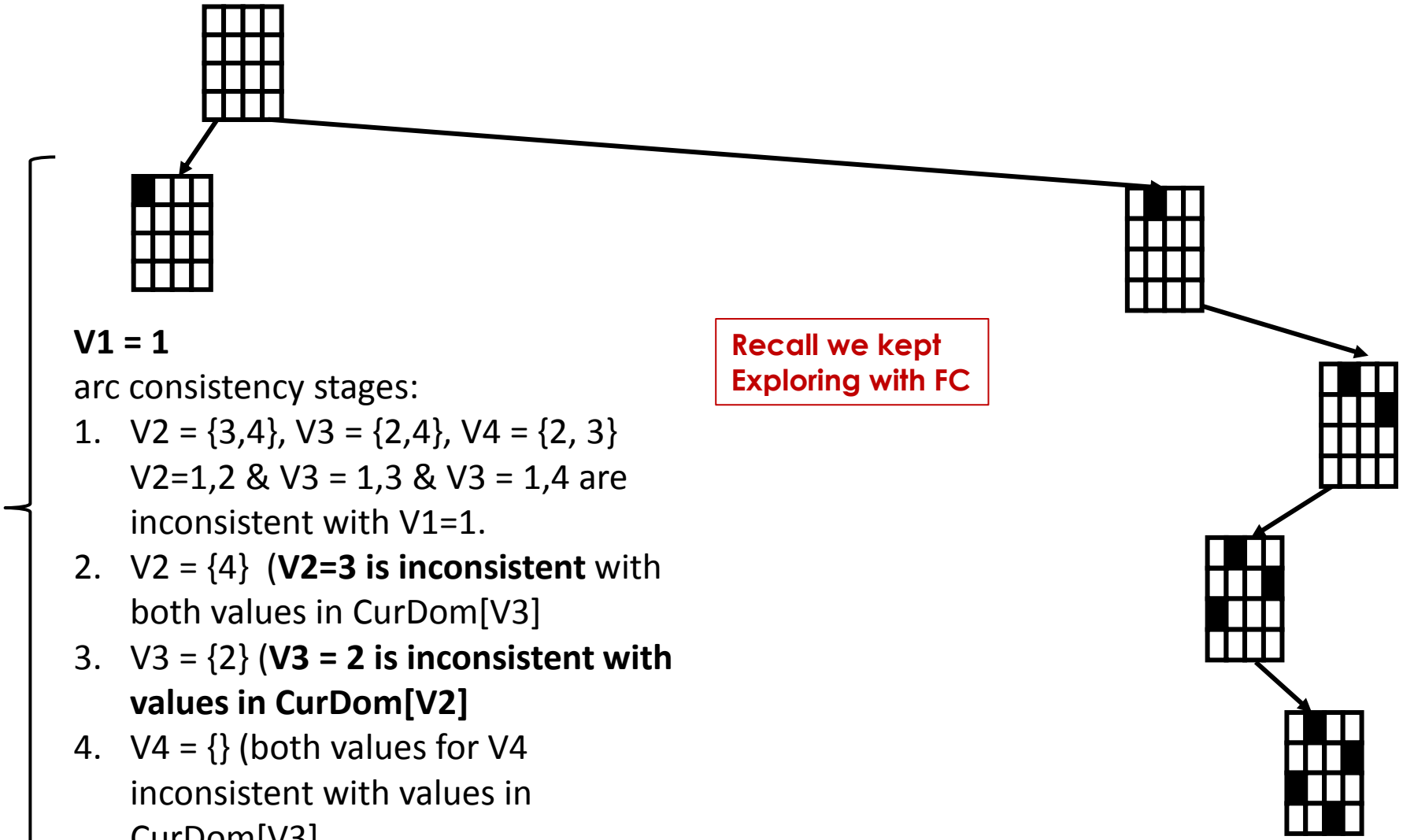
A simple example

Is this CSP arc consistent? If not, what values should be removed?



What about Dom[Y]=$\{2,4\}$?

Example: 4-Queens GAC search Space



V1 = 1

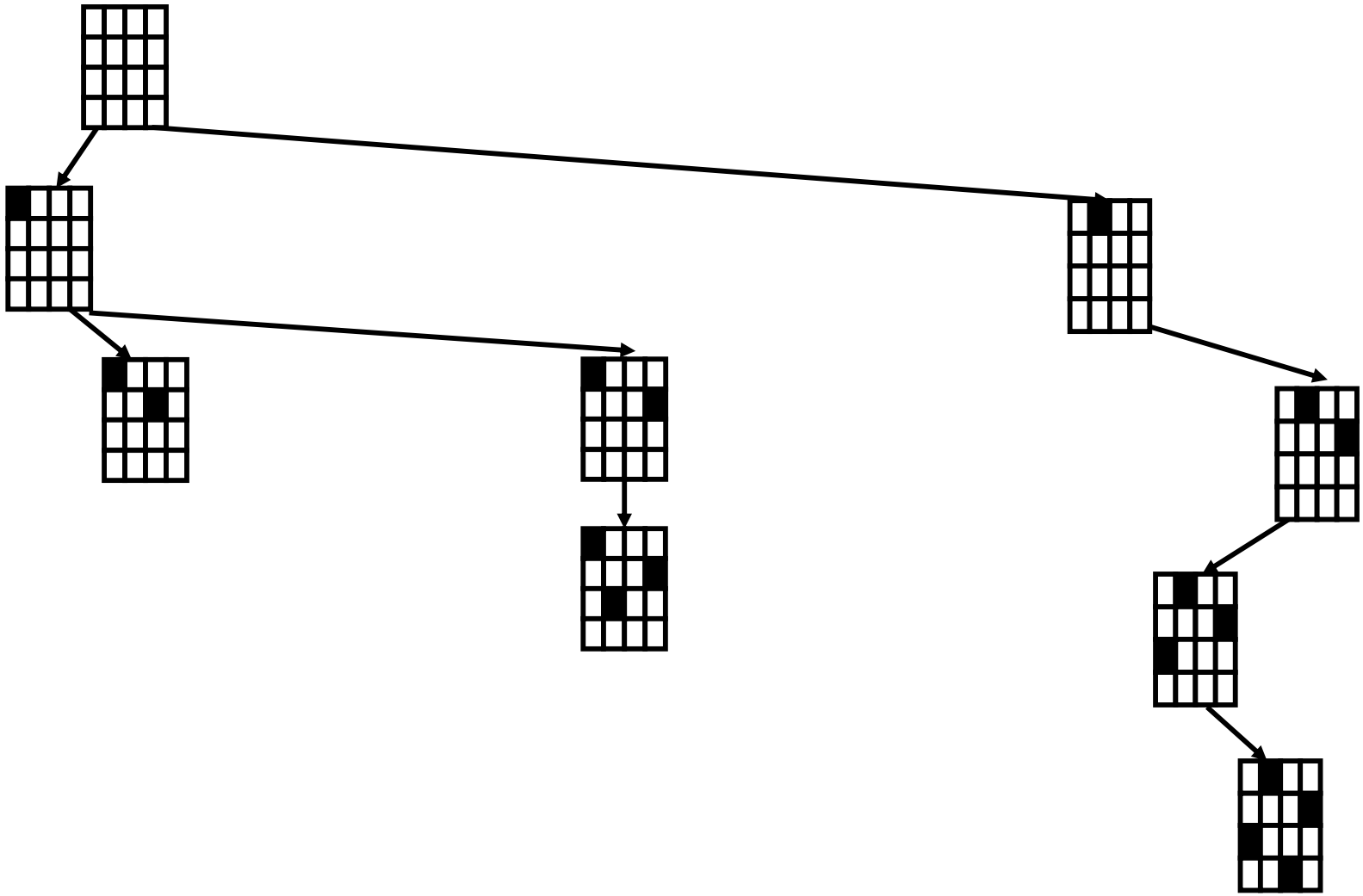
arc consistency stages:

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

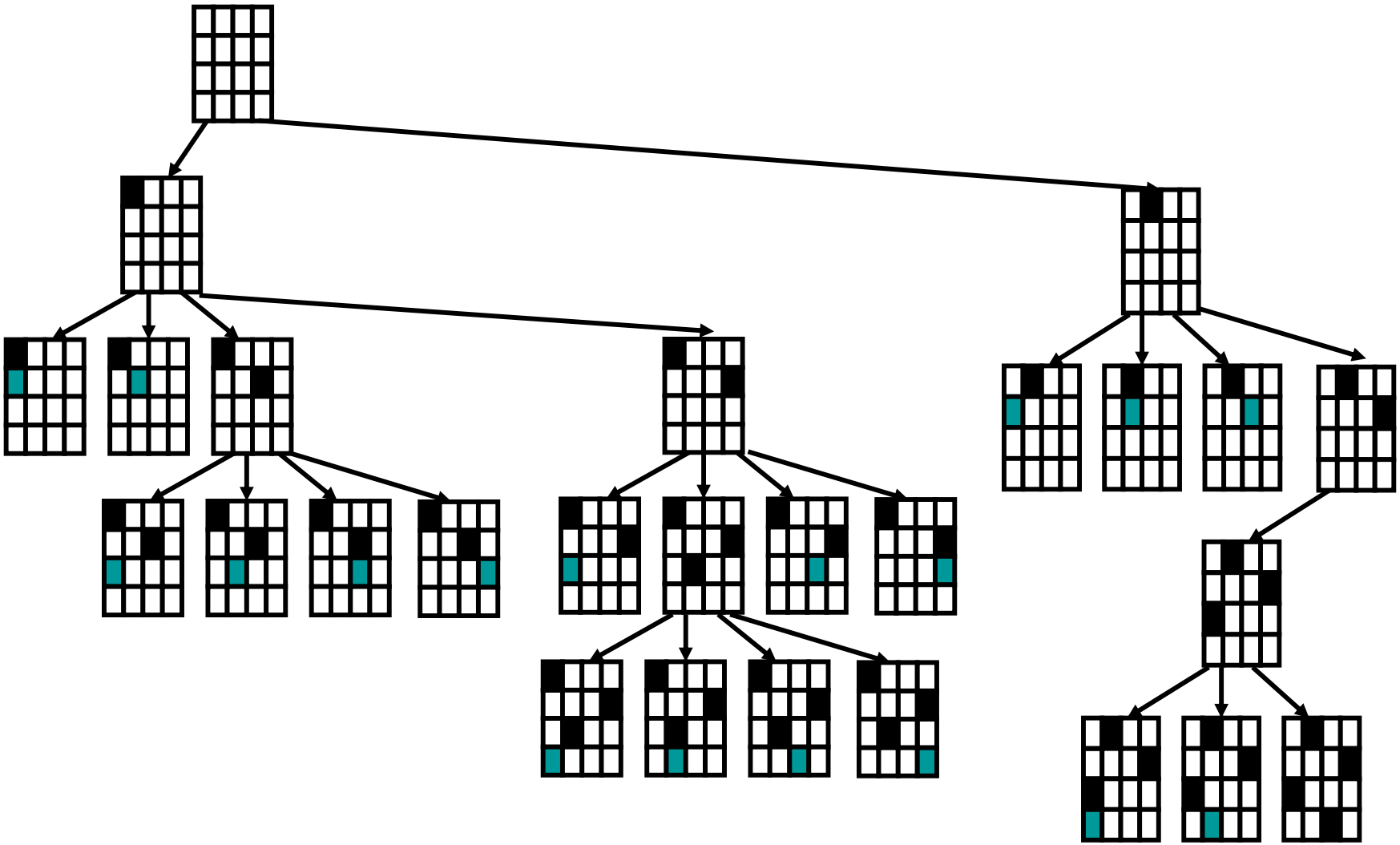
DWO

Recall we kept Exploring with FC

CONTRAST TO: 4-Queens FC search Space

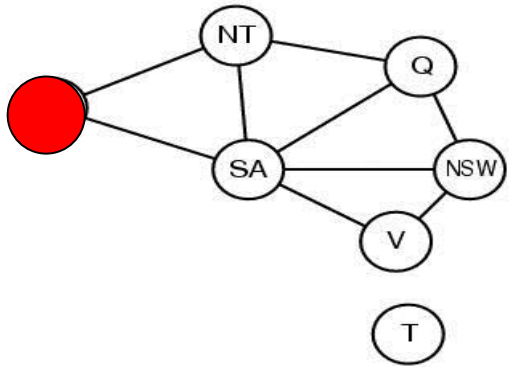


CONTRAST TO: 4-Queens Backtracking Search Space



Example – Map coloring

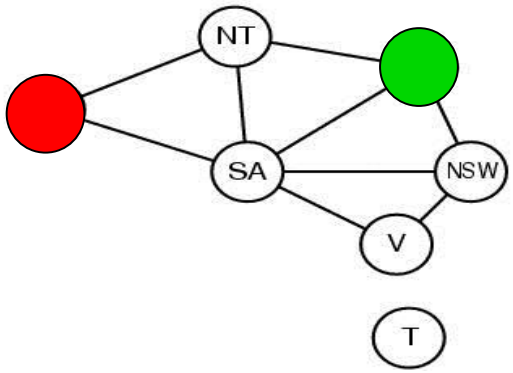
- Assign *{WA=red}*
- Effects on other variables connected by constraints to WA
  - *NT can no longer be red = {G, B}*
  - *SA can no longer be red = {G, B}*
- *All other values are arc-consistent*



Example – Map coloring

- 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.
- Like forward checking, GAC could also be performed before we even start to search, *i.e.*, before we assign any variables.
- 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
```

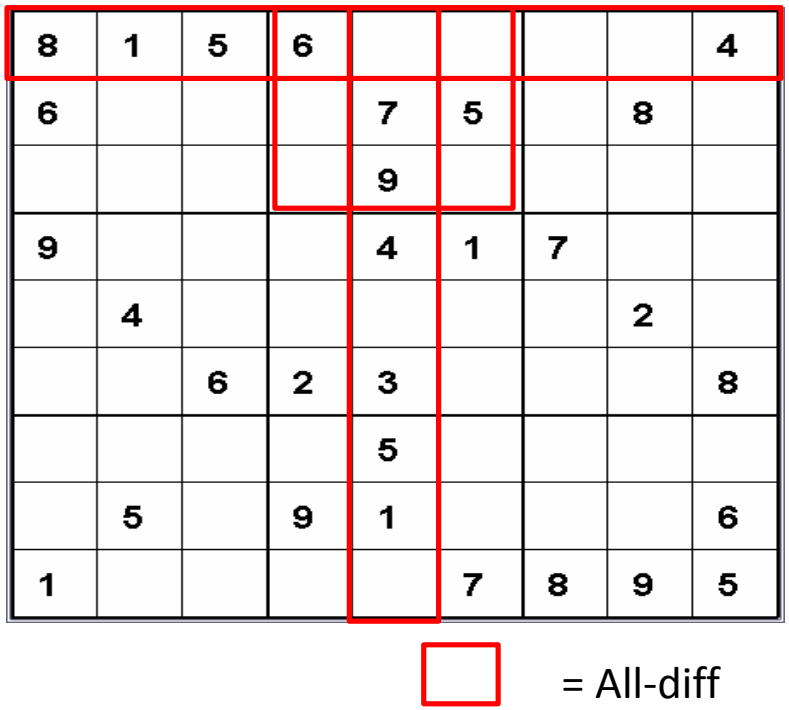
Enforce GAC

- A support for V=d in constraint C is an assignment A to all of the other variables in scope(C) s.t. A U {V=d} satisfies C.
- Smarter implementations keep track of "supports" to avoid having to search through all possible assignments to the other variables for a satisfying assignment.
- 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.
- Another key development in practice is that for some constraints this computation can be done in polynomial time.
- *e.g.*, all-diff$(V_1, \ldots, V_n)$ we can check if $V_i = d$ has a support in the current domains of the other variables in polynomial time using ideas from graph theory.

GAC enforce example



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.

Consider the case where we perform GAC inference *before* we do any search, limiting ourselves to consideration of only $C_{SS2,}$ C_{R1} and C_{C5} . We place these constraints on the GAC queue and call GAC enforce.

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_{C5}) ➔
CurDom of $V_{1,5}$ = {2}
CurDom of $V_{5,5}$,$V_{9,5}$ = {2, 6, 8}
GAC(C_{SS2}) ➔
CurDom of $V_{2,4}$, $V_{3,4}$, $V_{3,6}$ = {1, 4, 8}
GAC(C_{R1}) ➔
CurDom of $V_{1,6}$, = {3}
CurDom of $V_{1,7}$, $V_{1,8}$ = {7, 9}

Exercise: Forward checking

Consider the CSP with

- Variables $= \{V_1, \ldots, V_5\}$.
- $Dom[V_1] = \{1, 2, 3\}$, $Dom[V_2] = \{1, 2, 3\}$, $Dom[V_3] = \{1, 2, 3\}$, $Dom[V_4] = \{1, 2, 3, 4, 5\}$, $Dom[V_5] = \{1, 2, 3, 4\}$.
- Constraints: $V_i \neq V_j$, $i \neq j$.

Apply forward checking search to find the first solution.

- Instantiate the variables in the order V1, V2, V3, V4, V5.
- Assign the values in the current domain in increasing order.
- At each node indicate the variable being instantiated, and for each variable its updated CurDom.

GAC exercise

- $Dom[X] = Dom[Y] = Dom[Z] = \{1, 2, 3, 4\}$,
  $Dom[W] = \{1, 2, 3, 4, 5\}$
- $C_1 : X = Y + Z, C_2 : W > X, C_3 : W = X + Y + Z$
- Enforce GAC
- Use GAC algorithm to find the first solution

GAC example

- C1(V1,V2,V3)

V1	V2	V3
A	B	C
B	A	C
A	A	B

- C2(V1,V3,V4,V5)

V1	V3	V4	V5
A	A	A	A
A	B	C	B
B	C	B	B
C	A	B	C
C	B	A	B

- C3(V2,V3,V5)

V2	V3	V5
A	A	A
A	B	C
B	C	B
C	A	B
C	B	A

- Dom[V1]…Dom[V5] = {a, b, c}