Search 02
- Local Search
- Constraint Satisfaction Problem

PRAKARN UNACHAK

Outline

- Review
- Local Search
- Constraint Satisfaction Problem (CSP)

Review – Problem Formulation

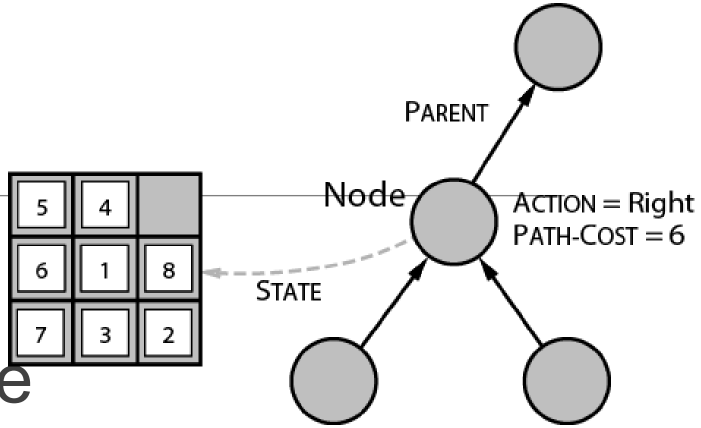
First, we translate problem into searchable form

1. Describe *goal* of the problem
2. Define *state* of the problem that can be searched
   - How many variables (if any) and what types?
   - The meaning of each variable
3. Define *initiate state*: where the search starts
4. Define actions that can be done, and when (which states) can they be performed

Review – Problem Formulation (cont.)

5. Define *goal test*: how to find out a state is a goal state
6. Define *transition model*: how state changes when an action is performed
7. If applicable, define *path cost* and *step cost*, cost of action on each state

Review – Tree Search



- At a state, we can perform some actions, each will take us to another (or the same) state
- Some problems want *path*: sequence of actions taken – best solution is solution with least *path cost*
- Some problems only want *configuration* (final state) that fit the description of goal
- For now, focusing on discrete *single-state problem*: *deterministic* and *fully-observable*

Review – Tree Search (cont.)

- Start at *root node* at initial state
  - Using *search strategy* to pick the next node in *frontier* to *expand*
    - Expand: perform actions that can be creating child nodes, put them in frontier
  - Stop when a node with goal state is select to expanded (<u>not created</u>)
- If we want to remember visited state in *explored set* – use *graph search*, prevent repetition

Review – Uninformed Search

- AKA blind search – search strategy only use information in problem definition
  - Do not approximate how “good” a non-goal state is
  - Can still use path cost

- *Breadth-first Search* (BFS): FIFO, expand shallowest node first, guarantee solution, optimal if step cost = 1, but expensive ($O(b^d)$) both in computational time and memory space

- *Depth-first Search* (DFS): LIFO, expand deepest node first, cheap (linear) in space, but still expensive in computational time and <u>does not</u> guarantee a solution

Review – Uninformed Search

- *Iterative Deepening Search* (IDS), use depth-limited search to perform DFS up to depth limited, iterativly increasing depth limit if no solution is found. Trade additional computational power for much cheaper memory space requirement
- *Uniform-cost Search* is use when step costs are not equal, expand node with least path cost. BFS with cost instead of depth. May expand many unnecessary nodes.

Review – Informed Search

- Approximate the path cost from current node to goal state, *heuristic function*, $h(n)$, - depend only on the state
  - Or, how "promising" a state is
  - Generally called *best-first search,* expand node with least *evaluation function,* $f(n)$ value, which can contain both path cost $g(n)$ and $h(n)$
  - Have space complexity issue as BFS
- Greedy best-first search: $f(n) = h(n)$
  - May not return optimal solution

Review – Informed Search (cont.)

- *A* search*: $f(n) = g(n) + h(n)$
  - Guarantee optimal solution if heuristic is
    - *Admissible* (never overestimate real cost) for tree search
    - *Consistent* (non-decreasing among search path) for graph search
  - Can be modified to handle space problem, IDA*, RBFS*, SMA*
- Admissible heuristic can be created by *relaxing* some restriction of the problem
  - One that *dominates* (greater value) the other is preferred – closer to real cost

Local Search

FOR WHEN EXACT SEARCH IS TOO COSTLY

When not to Use Tree Search?

- Tree search can be expensive, we might not have (or want to spend) enough resources
- And in some problems, we only care about getting state(s) that pass the goal test, and not the paths
  - State space is set of complete configurations (solutions) – most are not optimal
  - Examples: n-Queen, Crossword, Sudoku

Local Search

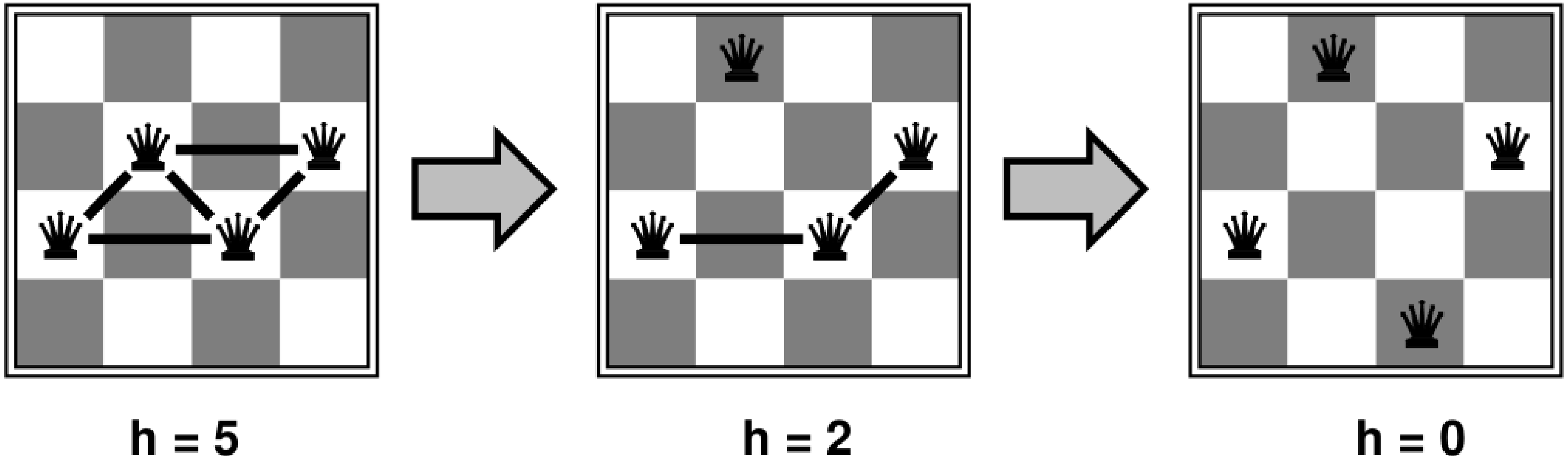
- We can use *local search*, or *iterative improvement algorithm*
  - Maintain one (or a set number of) node(s) and "improve" on them steps by steps
  - Constant Space
- Goal:
  - Optimal Configuration, or
  - Configuration that satisfy specified constraints, such as deadline

Optimization with Local Search

- Measure quality of solution with objective function
  - Example: using number of attacking pairs for n-Queen
  - Can be a minimization (reducing objective value) problem or maximization (increasing objective value) one
- The goal is to fine *global optimum* (maximum or minimum)
  - But local search cannot guarantee global optimal
  - Just “good enough” ones
- Local search can solve larger problems quickly

Example: n-Queens

- Start with complete configuration: one queen per column already placed
- Action: move queen vertically (row-wise) to reduce attacking pairs



- Evaluation function: heuristic cost *h* = number of attacking pairs

Hill-climbing search

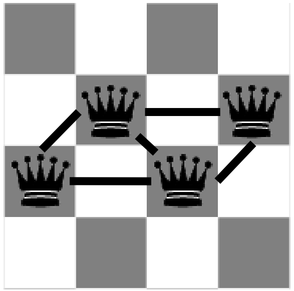
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    current ⟵ MAKE-NODE(problem.INITIAL-STATE)
    loop do
        neighbor ⟵ a highest-valued successor of current
        if neighbor.VALUE ≤ current.VALUE then return current.STATE
    current ⟵ neighbor
```

Stop when no better solution is found

- Or gradient ascent/descent, greedy local search
- Expand current node: create list of successors
  - If best successor (neighbor) is better than current, let that be current instead
- “Like an amnesiac climbing the Everest under thick fog”

Hill-climbing Search

- Example: 4-queen

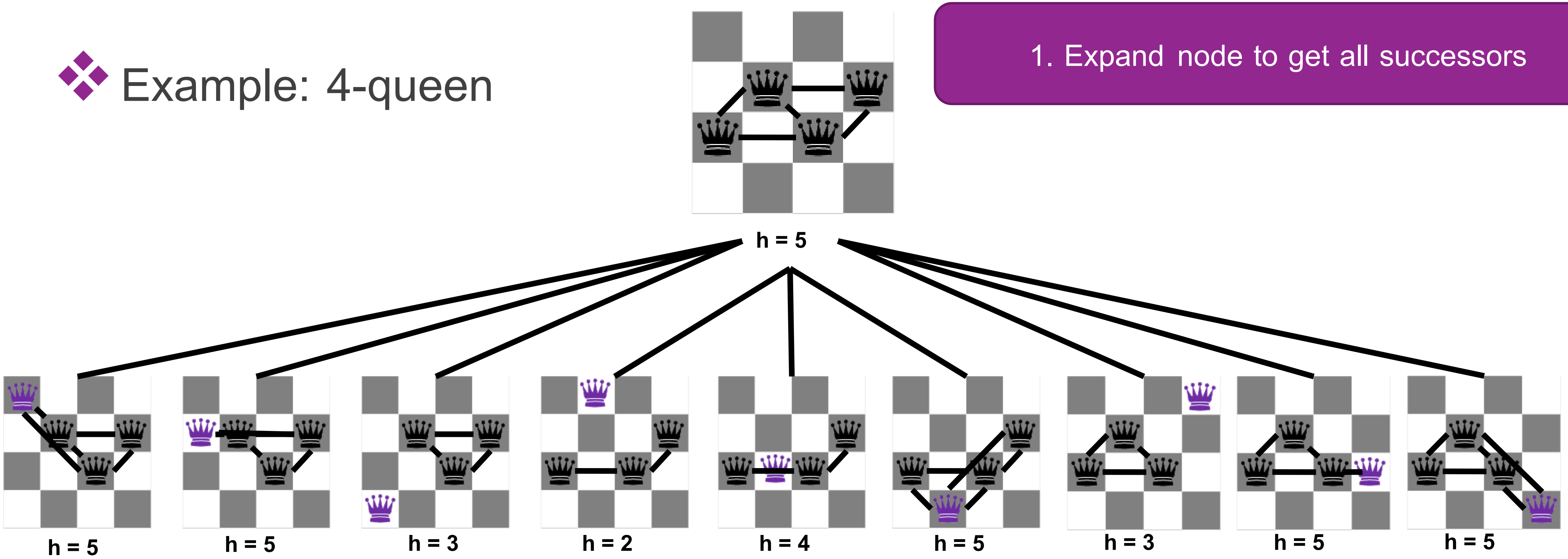


h = 5

Hill-climbing Search

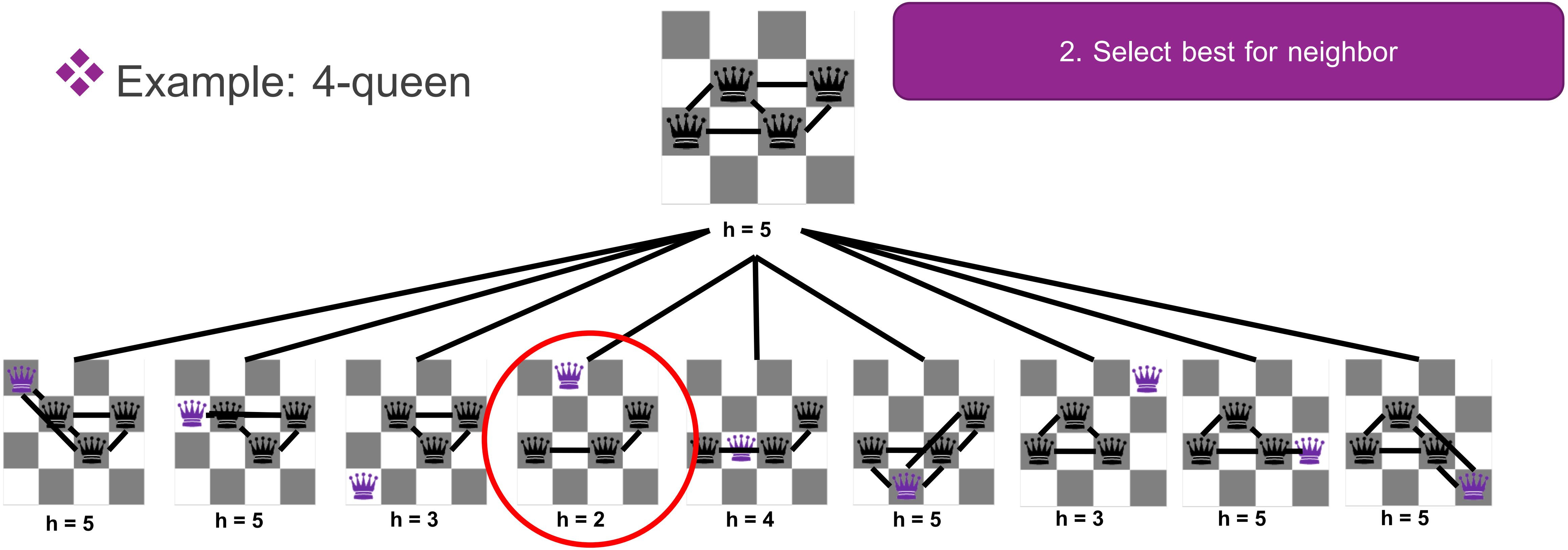
- Example: 4-queen

1. Expand node to get all successors



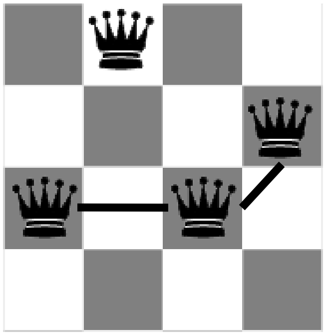
Hill-climbing search

- Example: 4-queen



Hill-climbing search

- Example: 4-queen

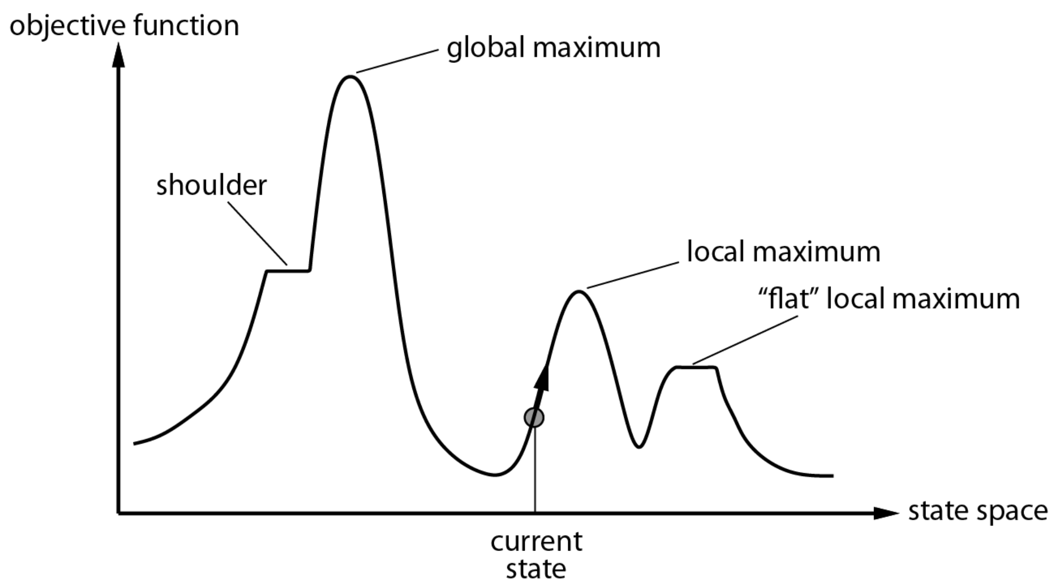


h = 2

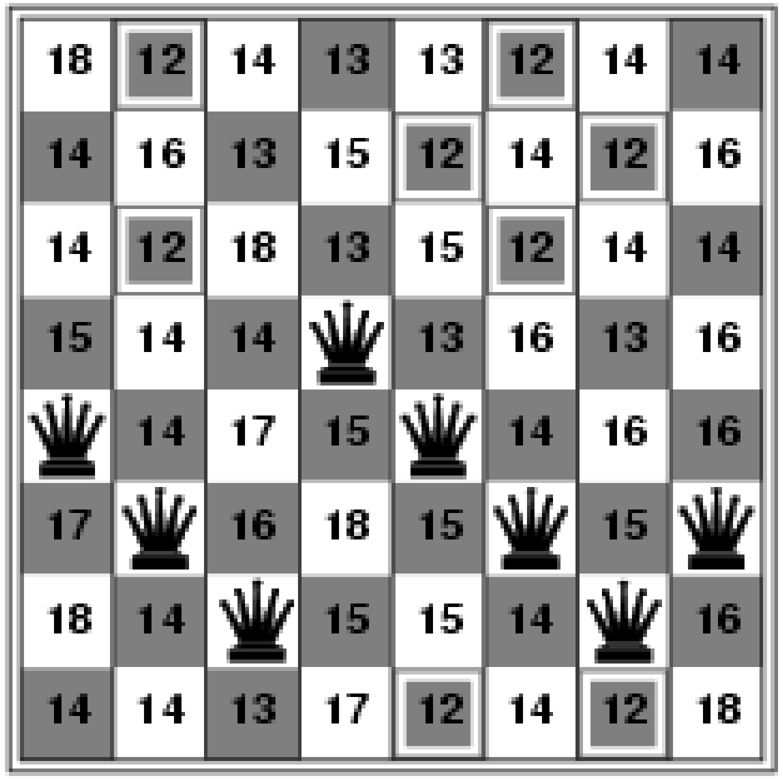
3. Neighbor is better than current, replace.
4. Repeat the search until no better neighbor

Problem with Hill-climbing search

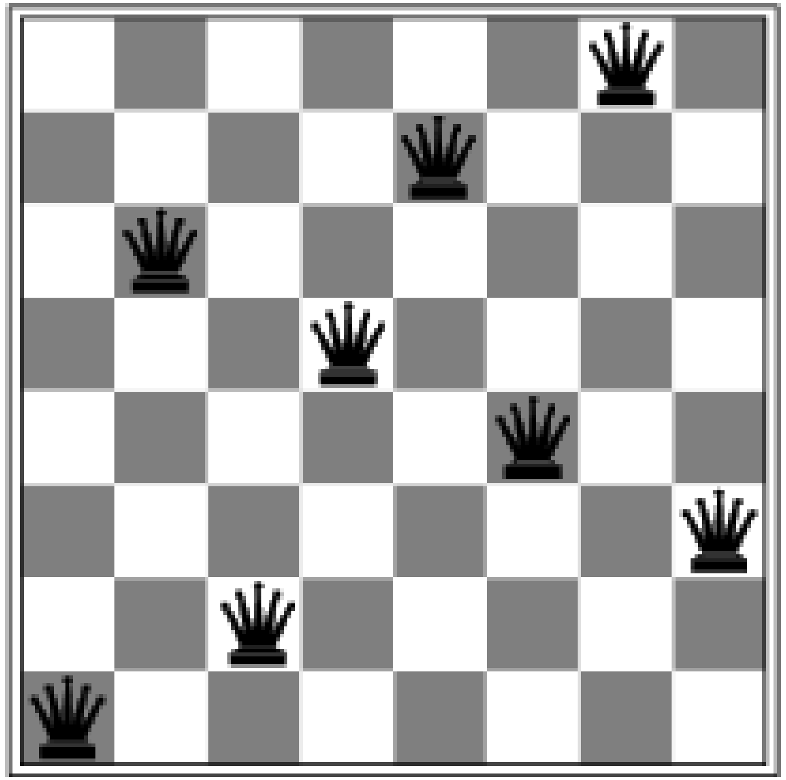
- Under certain Initial State, we can get stuck at a *local optimum* – a non-optimal solution that cannot be improved iteratively
- We need a way to exit local optimum:
  - *Random restart* – if stuck, randomly change current state
  - *Stochastic hillclimbing* – Allow "going downhill"



Example of Local Optima



h = *17*



Local minimum at *h* = *1*

Simulated Annealing Search

- A stochastic hillclimbing approach
- Annealing refers to the process of heating material (using metal) to change it properties
- Allow exiting local optima by letting worse state becoming current state, with reducing probability as the search goes on (hot $\longrightarrow$ cold)
- Use variable *T* (temperature) that will reduce its value over time, according to temperature schedule
  - Stop searching when *T* reaches 0
  - Slow enough temperature schedule will make probability of finding optimal solution reaching 1

Simulated Annealing Search (cont.) – Maximization

function SIMULATED-ANNEALING(*problem*, *schedule*) returns a solution state

inputs: *problem*, a problem

schedule, a mapping from time to "temperature"

current ⟵ MAKE-NODE(*problem*.INITIAL-STATE)

for *t* = 1 **to** ∞ **do**

$T \longleftarrow$ *schedule*(*t*)

if $T = 0$ **then return** *current*

next ⟵ a randomly selected successor of *current*

$\Delta E \longleftarrow$ *next*.VALUE – *current*.VALUE

if $\Delta E > 0$ **then** *current* ⟵ *next*

else *current* ⟵ *next* only with probability $e^{\Delta E/T}$

Tabu Search

- Like simulated annealing, allowing going to worse state if no better one can be found
- But maintain tabu list: list of recently-visited states
  - Will not expand state in tabu list
  - Tabu list have limited size, will remove oldest if full
- Need separate *stopping condition* to end the search
  - Number of iterations, evaluation value goal, etc.

```
function TABU-SEARCH(problem, stoppingCondition, maxTabuSize)
returns a solution state
    inputs:     problem, a problem
                    stoppingCondition, for checking when to stop searching
                    maxTabuSize, maximum tabu size
    current ⟵ MAKE-NODE(problem.INITIAL-STATE)
    tabuList ⟵ {}
    while (not stoppingCondition()) do
        neighbor = a successor of current
        for i in successors(current)
            if (not tabuList.contains(i)) and ( i.VALUE  > neighbor.VALUE) then neighbor ⟵ i
        if neighbor.VALUE > current.VALUE then current ⟵ neighbor
        else
                if (tabuList.size() >= maxTabuSize )       then
                    tabuList.popOldest()
                current ⟵ neighbor
        tabuList.push(neighbor)
    return current
```

Other Local Search – Beam Search

- Beam Search
  - Maintain multiple (*k*) current nodes
  - In each step, create successors of all *k* nodes, stop if goal/optimal solution is reach. If not, select *k* nodes among successor and current to be new current nodes

Other Local Search – Genetic Algorithm

Genetic Algorithm

- Similar to beam search, where multiple current states are maintained in *population*
- But solutions are represented in *chromosome* form, usually arrays
- Genetic operators (crossover and mutation) are used to created successors
- Only fittest solutions will survive to next iteration (generation), maintaining population size and improving solutions

Constraint Satisfaction Problem (CSP)

HOW TO ELIMINATE UNPRODUCTIVE CHOICE AS SOON AS POSSIBLE

Problems and Constraints

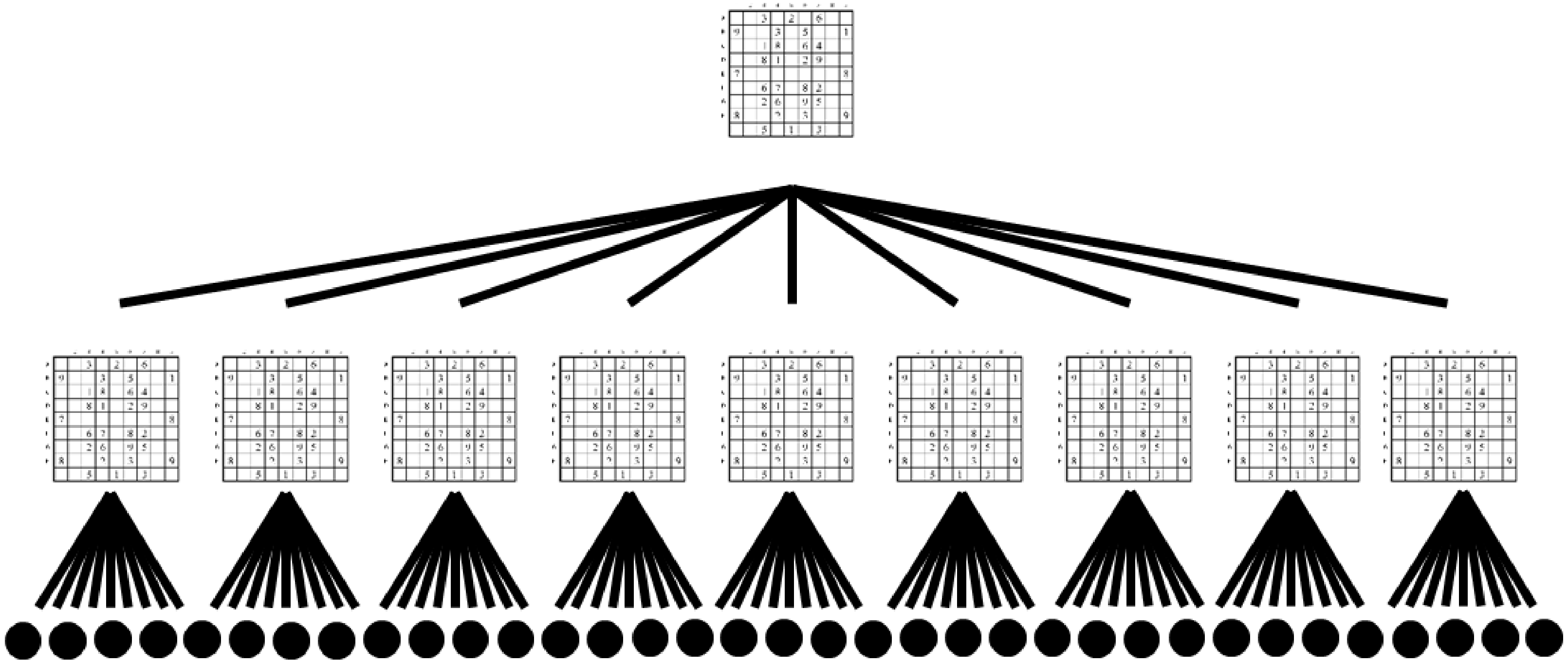
- In some problems, the goal is to find a complete configuration that satisfy all specified constraints

- Example: Sudoku – fill number 1-9 to each square such that:
  - Each row has all 9 numbers
  - Each column has all 9 numbers
  - Each 3x3 group has all 9 numbers

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

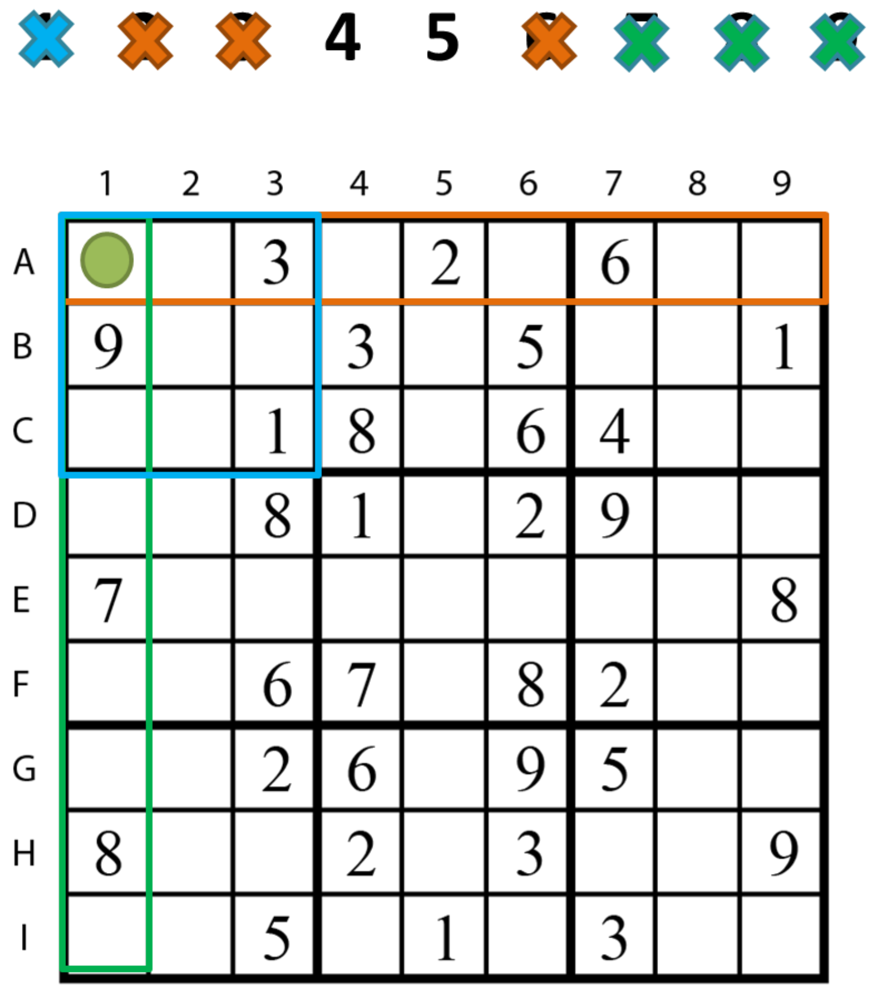
Cursing of Branching Factor (*b*)

- If we define action to be filling in number to a square.
  - If we have 49 empty squares, we can have $9^{49} = 5.7264169 \times 10^{46}$ nodes!



Solving Problem using Constraints

- However, not all choices will lead to acceptable solutions
  - Constraints can be violated
- Example, at square A1, we need to check numbers at
  - Row A
  - Column 1
  - And group A1-C3
- This can reduce larger number of choices to be considered, and can be reduced further as search go on



Constraint Satisfaction Problems (CSP)

CSP:

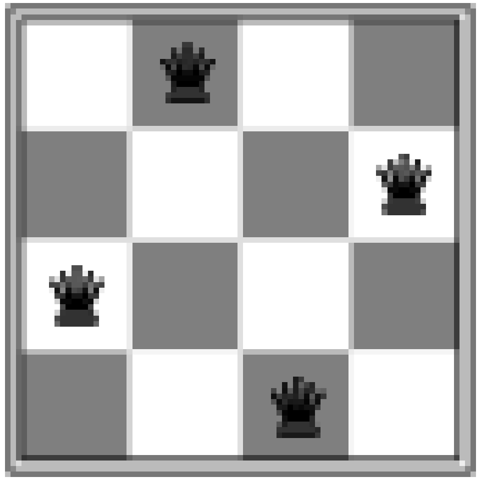
- These problem can be represented in *factored representation*
  - State consists of multiple variables
- A valid solution is found when all variables have values that satisfy all constraints in the problem

Constraint Satisfaction Problems (cont.)

- CSP consists of 3 sets: *X*, *D* and *C*
  - *X* is set variables $\{X_1,\ldots,X_n\}$ representing the state
  - $D = \{D_1,\ldots,D_n\}$ where D_i is domain of X_i, set of its possible values
  - *C* is set of constraints that specify allowable combinations of values
- Goal is to find assigned values of D_i in X_i for all variable in *X* that **does not violate any constraint in *C*** – *Constraint Consistent*
- We will show a general-purpose algorithm for all CSPs

Example: 4-queens

- States: 4 queens in 4 columns
  - 4^4 = 256 possible states
  - X_i: Row location of each queen
  - D_i: 1,2,3,4 for all i
  - C: No attacking pairs (how many pairwise constraints?)



1. cannotAttack(X_1, X_2)
2. cannotAttack(X_1, X_3)
3. cannotAttack(X_1, X_4)
4. cannotAttack(X_2, X_3)
5. cannotAttack(X_2, X_4)
6. cannotAttack(X_3, X_4)

Example: Map-coloring

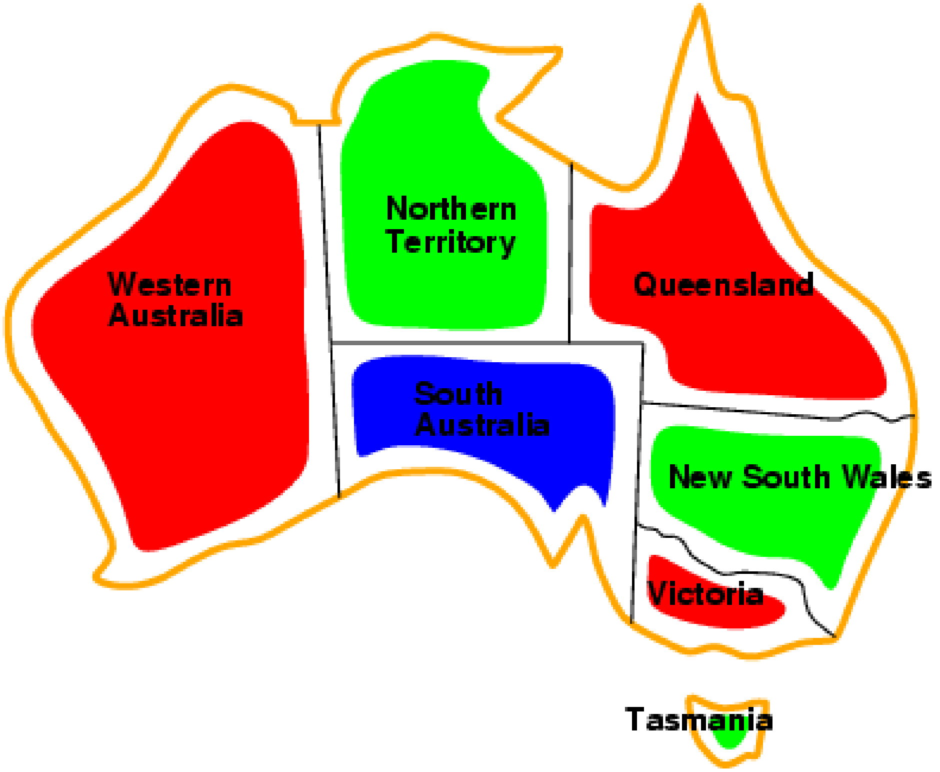
- Color each state of Australia in the map using {red, green, blue} such that **no states that share border have the same color**
- *X*: *WA, NT, Q, NSW, V, SA, T*
- D_i: {red, green, blue}
- *C*: Border-sharing states need to use different color

{SA ≠ WA, SA ≠ NT, SA ≠ Q, SA ≠ NSW , SA ≠ V, WA ≠ NT, NT ≠ Q, Q ≠ NSW , NSW ≠ V }

- Example: *WA* ≠ *NT*, หรือ (*WA*, *NT*) Have to be from set {(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)}

Map-coloring (cont.)

- Solution is assigning values from domain to all variables (*complete*) and satisfy all constraints (*consistent*)
- Example: WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green



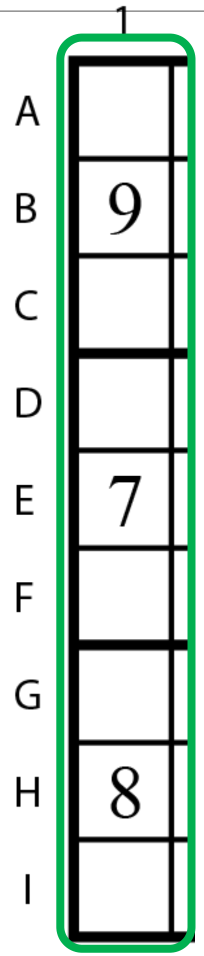
Example: Sudoku

- X = ?
- D_i = ?
- C = ?

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Constraint can be between more than two variables

- Using only constraints between one or two variables can create large amount of constraints
- Can define constraints between 3+ variables for convenient



1. $A1 \neq 9$
2. $A1 \neq C1$
3. $A1 \neq D1$
4. $A1 \neq 7$
5. $A1 \neq F1$
6. $A1 \neq G1$
7. $A1 \neq 8$
8. $A1 \neq I1$
9. $B1 \neq C1$

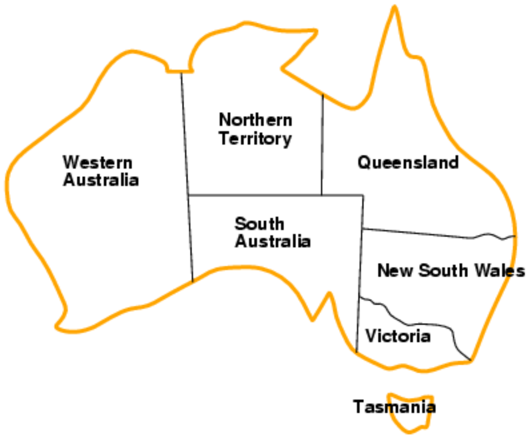
….

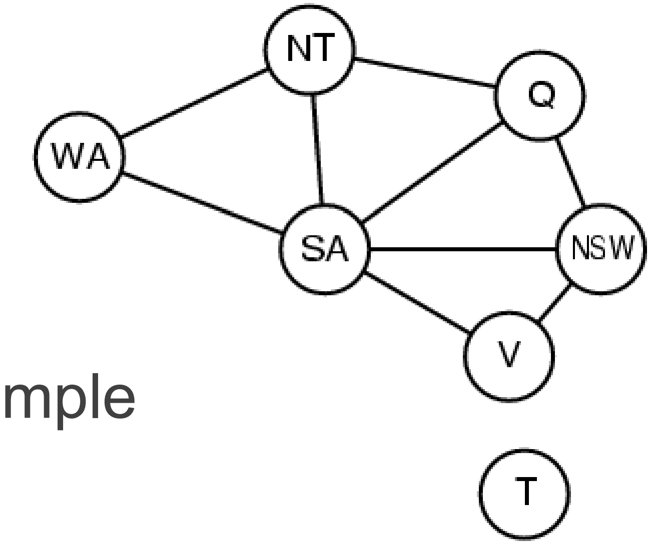
36. $H1 \neq I1$

Alldiff(A1, 9, C1, D1, 7, F1, G1, 8, I1)

Constraint Graph

- Used to show constraints between variable
- In Constraint Graph
  - Nodes are variables
  - Edges (Arc, Link) connect variables that share constraints
- Can use graph structure to speed the search
  - Can consider Tasmania as separate problem, for example





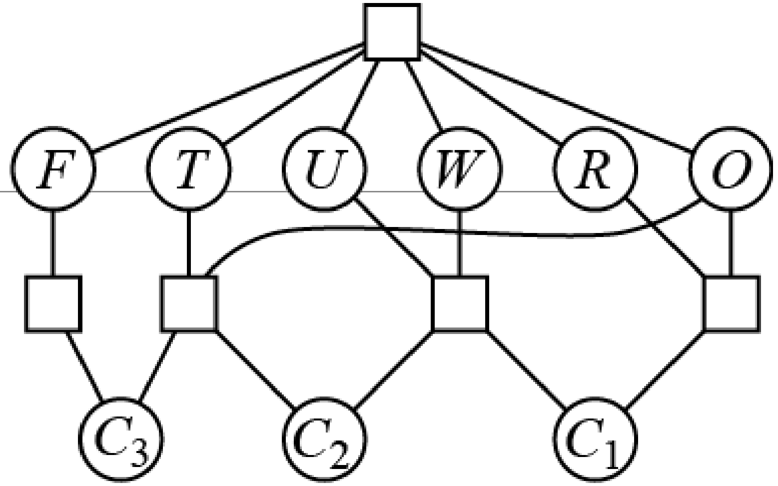
Type of Constraints

- Unary: involves 1 variable
  - Example: SA ≠ green
- Binary: involves 2 variables
  - Example: SA ≠ WA
  - Constraint can be directional: even if there is constraint {*X*, *Y*} does not mean there's also constraint {*Y*, *X*}
- Global: involves 3+ variables
  - Example: Alldiff()
  - Constraint graph becomes *constraint hypergraph*

ตัวอย่าง: Cryptarithmetic

$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$

(a)



(b)

- Variables: $F\ T\ U\ W\ R\ O\ C_1\ C_2\ C_3$
- Domains:
  - {0,1,2,3,4,5,6,7,8,9} for *F, T, U, W, R,* and *O*
  - *{0, 1} for* carry variables C_1, C_2, and C_3
- Constraints: *Alldiff* (*F,T,U,W,R,O*)
  - $O + O = R + 10 \cdot C_1$
  - $C_1 + W + W = U + 10 \cdot C_2$
  - $C_2 + T + T = O + 10 \cdot C_3$
  - $C_3 = F$, $T \neq 0$, $F \neq 0$
- Constraints between variable shown in constraint hypergraph (b)

Real-world CSPs

- Assignment Problems
- Timetabling Problems
- Transportation/Factory Scheduling

- Can contain real-valued (continuous) variable
  - May need other technique, such as linear programming
- Can also contain preference (soft) constraints
  - Not required to satisfy, but the valid solution that does is consider better

Constraint propagation

INFERENCE WITH CSPS

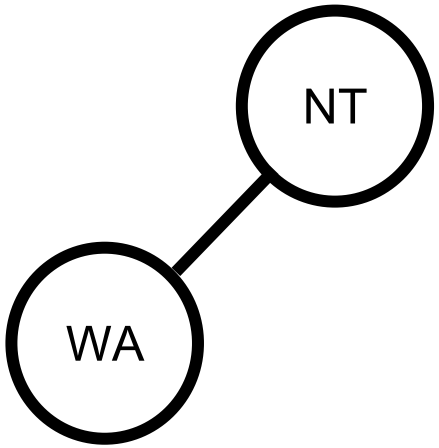
Constraint Propagation

- An inference technique
  - Creating new knowledge from available knowledge
- In this case, using constraints to reduce number of choices (variable values) to consider
- We need local consistency: consistent in node/arc/path/groups
  - Constraints as shown on hypergraph

Local Consistency

For each variable

- **Node Consistency**
  - Remaining values in variable's domain satisfy **unary** constraint(s) of that variable
- **Arc Consistency**
  - Remaining values in variable's domain satisfy **binary** constraint(s) that variable is involved in
  - Generalized arc-consistent is for **n-ary** constraints



Local Consistency (cont.)

Path Consistency

- Involving third variables in binary constraints
- $\{X_i, X_j\}$ is path-consistent to X_m when:
  - For every $\{X_i = a, X_j = b\}$ that satisfy all constraints on $\{X_i, X_j\}$, there will be value for X_m that satisfy all constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$

k-consistency

- Path consistency when expanded from k-1 to k variables

Global Constraints Consistency

- ex. *Alldiff*()
- Simple Algorithm:
  1. Assign value to variable with singleton domain (one remaining value) and remove them from consideration
  2. Remove values form domain of other variables that will not satisfy shared constraints with variable from 1.
  3. Repeat until (1) no variables with singleton domain remain, or (2) there is a variable with empty domain $\rightarrow$ inconsistency

Global Constraints Consistency (cont.)

- Example: Sudoku

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Constraint Propagation in Sudoku

Square E6

- Before:

  $D_{E6} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

- Column Constraint *Alldiff(A6,..,I6)*

  $D_{E6} = \{1, \cancel{2}, \cancel{3}, 4, \cancel{5}, \cancel{6}, 7, \cancel{8}, \cancel{9}\}$

- Row Constraint *Alldiff(E1,..,E9)*

  $D_{E6} = \{1, \cancel{2}, \cancel{3}, 4, \cancel{5}, \cancel{6}, \cancel{7}, \cancel{8}, \cancel{9}\}$

- Group Constraint *Alldiff(D4,..,D6,E4,..,E6,F4,..,F6)*

  $D_{E6} = \{\cancel{1}, \cancel{2}, \cancel{3}, 4, \cancel{5}, \cancel{6}, \cancel{7}, \cancel{8}, \cancel{9}\}$

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7					?			8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Global Constraints (cont.) – Additional Issues

- Resource/Production constraints
  - Sum of variable values (resource assigned) must not exceed available resource OR
  - Sum of variable values (production) must meet quota
- Can use bound propagation
  - Domain is range of number
  - Use constraint to reduce the range of domain
- CSP solution will be Bound Consistent when, for both upper and lower bound of X's values there are values for Y that satisfy constraints between X and Y

Example

F1, F2 are number of passengers on two planes

D1 = [0, 165]

D2 = [0, 385]

C = {F1+F2 = 420}

↓

D'1 = [35, 165]

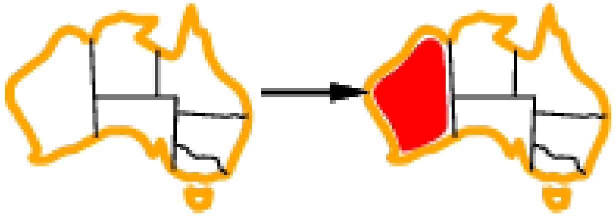
D'2 = [255, 385]

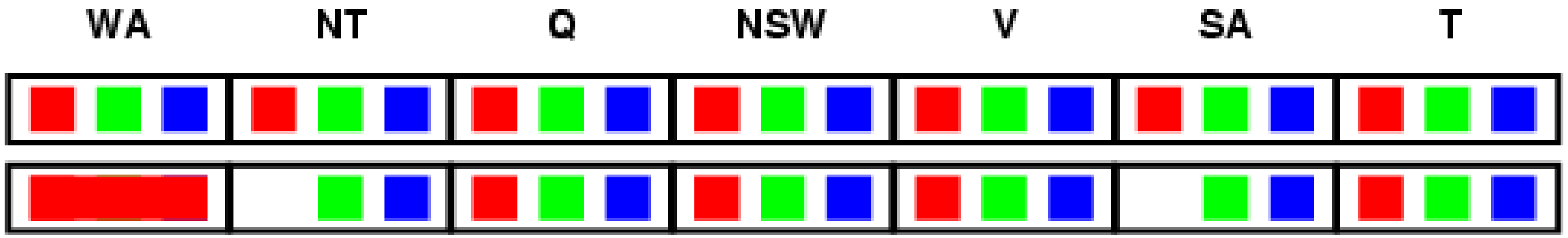
Inference with CSP

- Reducing Choices using Constraints
- Forward-checking
  - When a variable X is assigned, check arc-consistency between X and variable with constraints with X
- Constraint Propagation
  - When a variable is assigned OR when the domain of a variable is change, check arc-consistency and update domain of related variable
  - Repeat until no change in domains, or empty domain detected

Inference: Forward checking

- Remember domains of unassigned variables
- Stop search when a domain becomes empty





- Assigned *WA*
- Check Arc Consistency of (*WA*, *NT*) and (*WA*, *SA*) and adjust domain of *NT* and *SA*

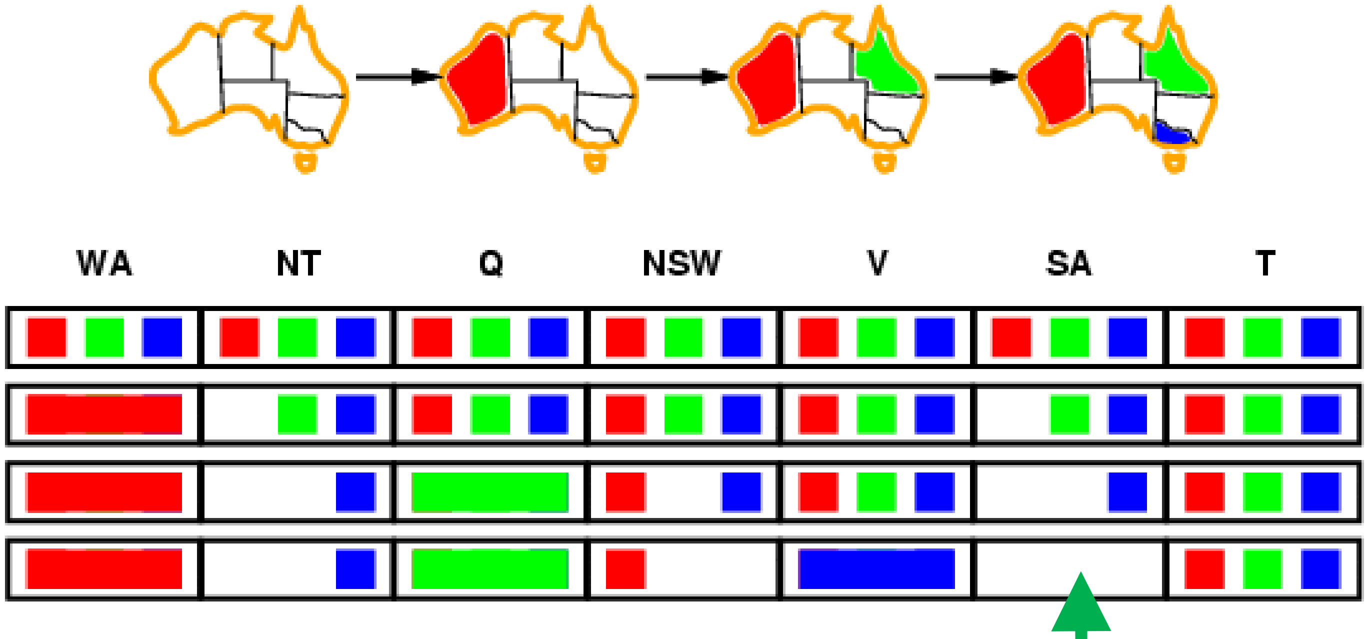
Inference: Forward checking

- Remember domains of unassigned variables
- Stop search when a domain becomes empty

- Assigned *Q*
- Check arc consistency of (*Q*, *NT*), (*Q*, *SA*) and (*Q*, NSW) and adjust domain of *NT*, *NSW* and *SA*

Inference: Forward checking

- Remember domains of unassigned variables
- Stop search when a domain becomes empty



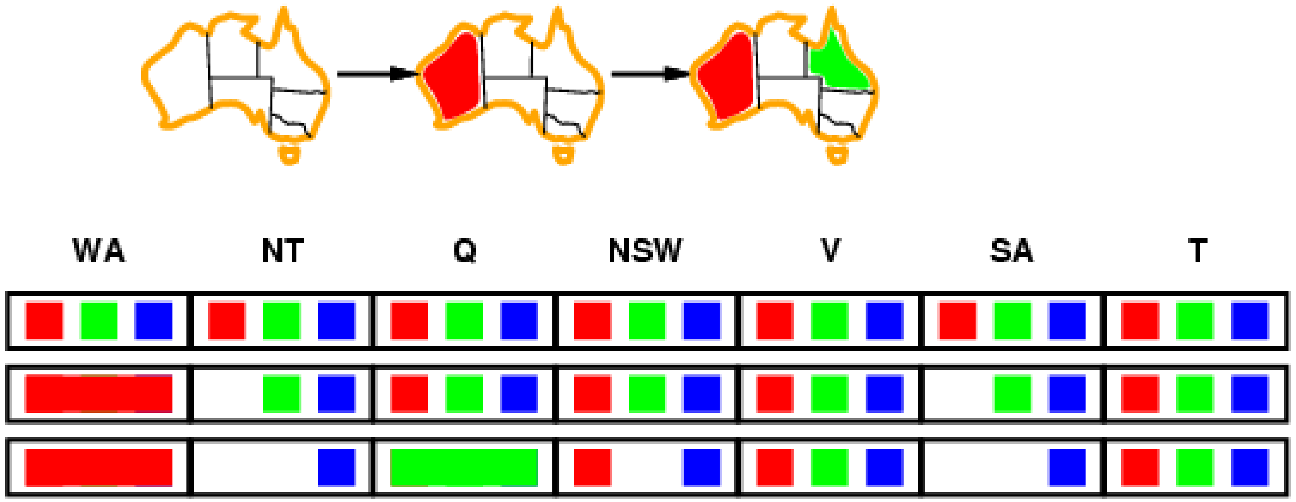
SA has empty domain

→ Stop search

- Assigned *V*...

Inference: Constraint propagation

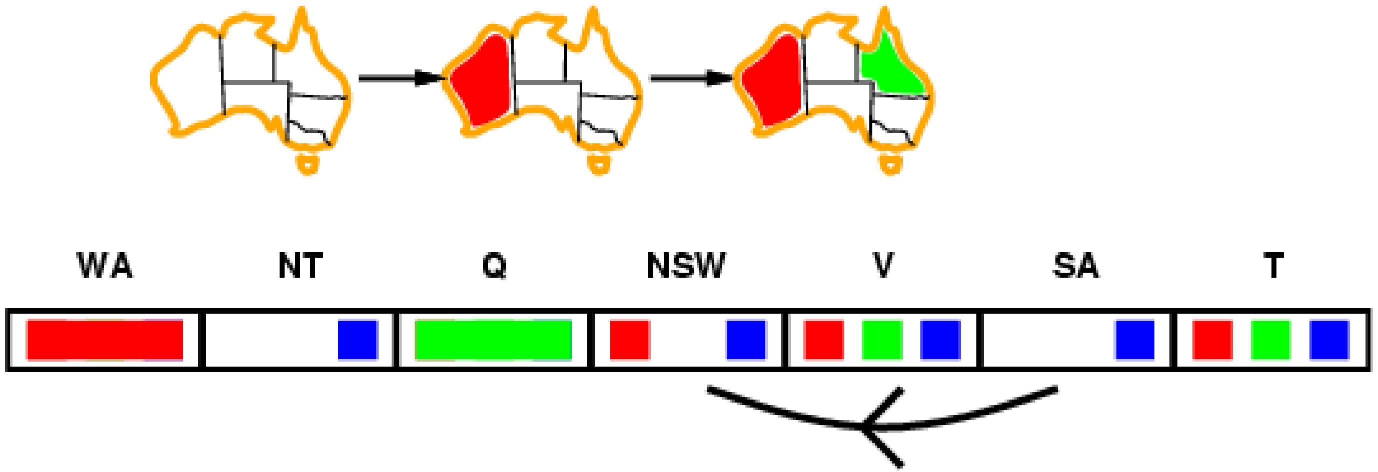
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures::



- NT and SA cannot both be blue
- Constraint propagation repeatedly enforces constraints locally
- Can use AC-3 algorithm

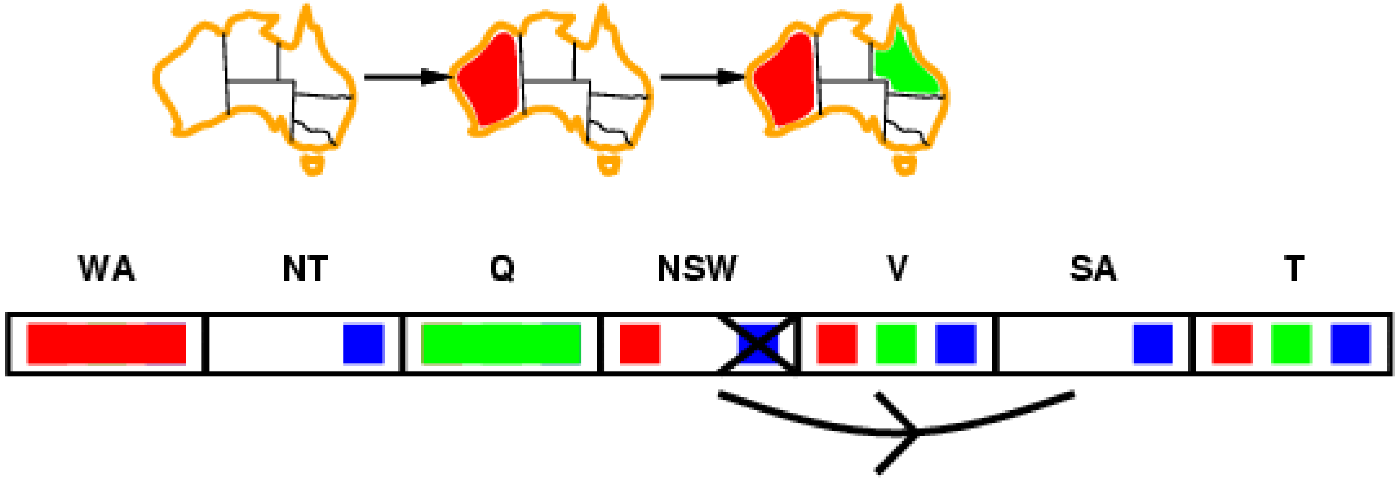
Arc consistency

- From constraint graph
- Simplest form of propagation makes each arc consistent
- *X* → *Y* is Consistent if and only if
  - for every value x of X there is some allowed y



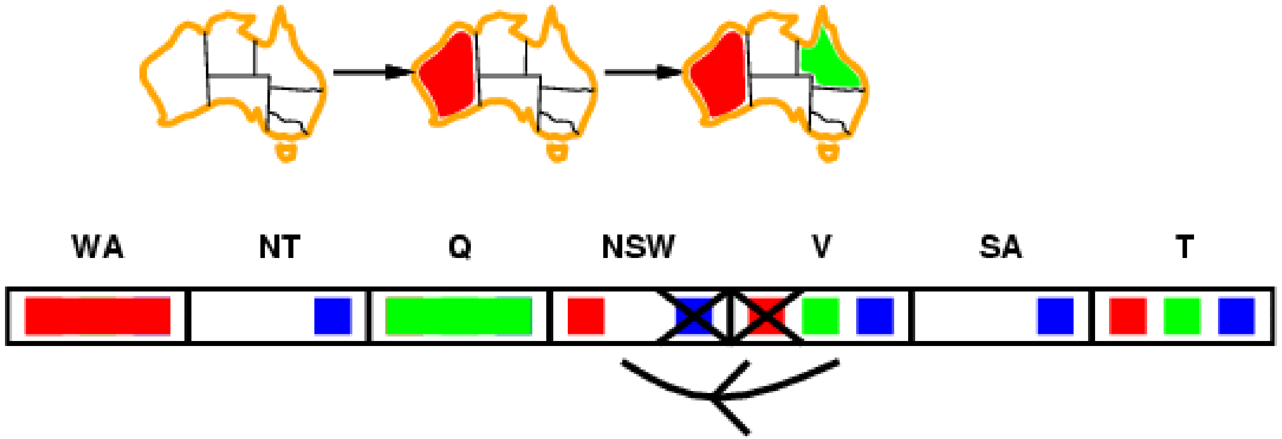
Arc consistency (AC-3)

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is Consistent if and only if
  - for every value x of X there is some allowed y



Arc consistency (AC-3)

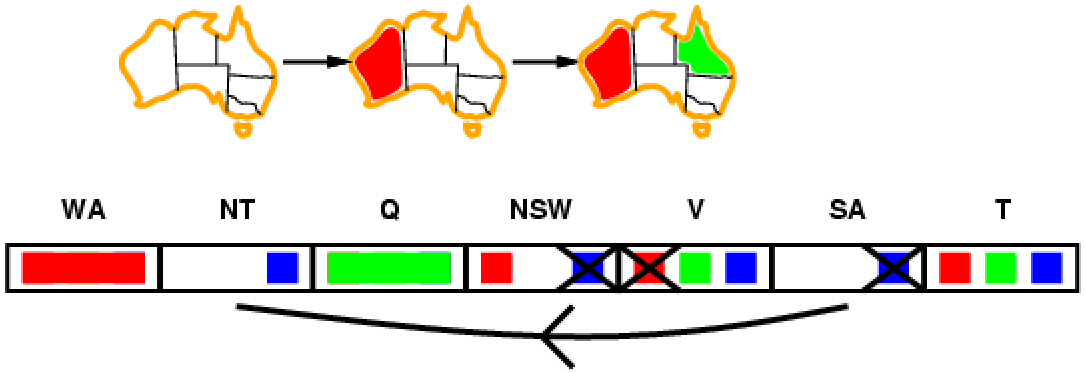
- Simplest form of propagation makes each arc consistent
- *X* → *Y* is Consistent if and only if
  - for every value x of X there is some allowed y



- If *X* loses a value (domain updated), all nodes with shared constraints to *X* need to be rechecked

Arc consistency (AC-3)

- Simplest form of propagation makes each arc consistent
- *X* → *Y* is Consistent if and only if
  - for every value x of X there is some allowed y



- If *X* loses a value (domain updated), all nodes with shared constraints to *X* need to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a <u>preprocessor</u> or <u>after each assignment</u>

AC-3 Algorithm

function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise

inputs: *csp*, a binary CSP with components (*X*, *D*, *C*)

local variables: *queue*, a queue of arcs, initially all the arcs in csp

while *queue* is not empty **do**

(*Xi* , *Xj*) ⟵ REMOVE-FIRST(*queue*)

if REVISE(*csp*, *Xi* , *Xj*) **then**

if *size* of *Di* = 0 **then return** false

for each *Xk* in *Xi*.NEIGHBORS - {*Xj*} do

add (*Xk*, *Xi*) to *queue*

return true

Time complexity: $O(n^2d^3)$

n — Number of Variables

d — Domain size

function REVISE(*csp*, *Xi* , *Xj*) **returns** true if we revise the domain of Xi

revised ⟵ *false*

for each *x* **in** *Di* **do**

if no value *y* in *Dj* allows (*x* ,*y*) to satisfy the constraint between *Xi* and *Xj* **then**

delete *x* from *Di*

revised ⟵ true

return *revised*

SEARCHING: Backtracking in CSPs

Standard Search Formulation (incremental)

Start with straightforward approach (tree search), then fix (backtracking)

States are defined by the values assigned so far

- Initial State: empty assignment { }
- Successor Function: assign a value to an unassigned variable **that does not conflict** with current assignment → fail if no legal assignment (empty domain)
- Goal Test: complete assignment

- Works with all CSPs
- Every solution will appear depth n with n variables → Use DFS
- Path is not important, can use complete-state formulation
- $b = (n - \ell)d$ at depth ℓ, therefore, we will (can) have $n! \cdot d^n$ leaves

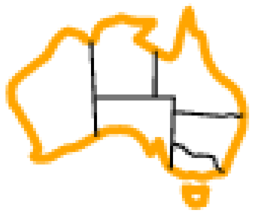
Backtracking search

- Variable assignments are commutative: order of assignment is not important, for example:

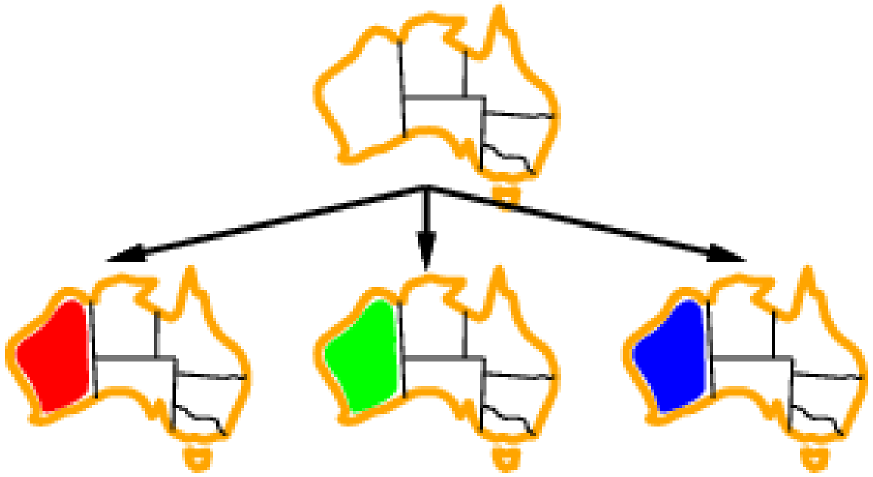
[WA = red **then** NT = green] is the same as [NT = green **then** WA = red]

- Only need to consider assignments to a single variable at each node → $b = d$ and there are d^n leaves
- Depth-first Search for CSPs with single-variable assignments is called backtracking search
- Backtracking search is the basic uninformed algorithm for CSPs
- Can solve *n*-queens at $n \approx 25$

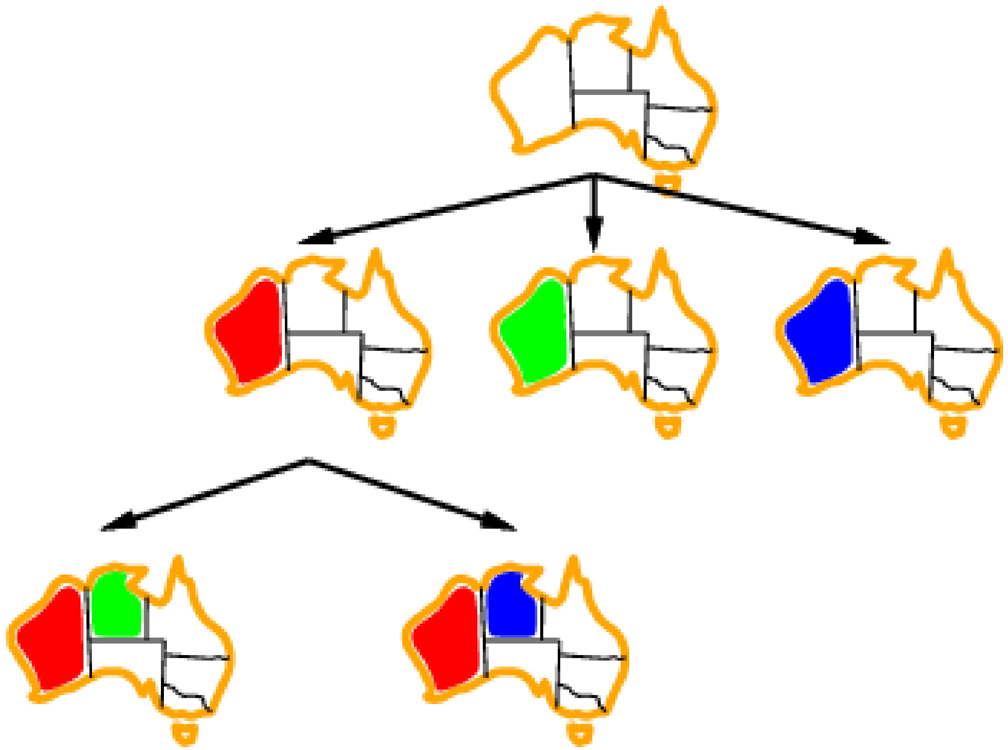
Backtracking Example



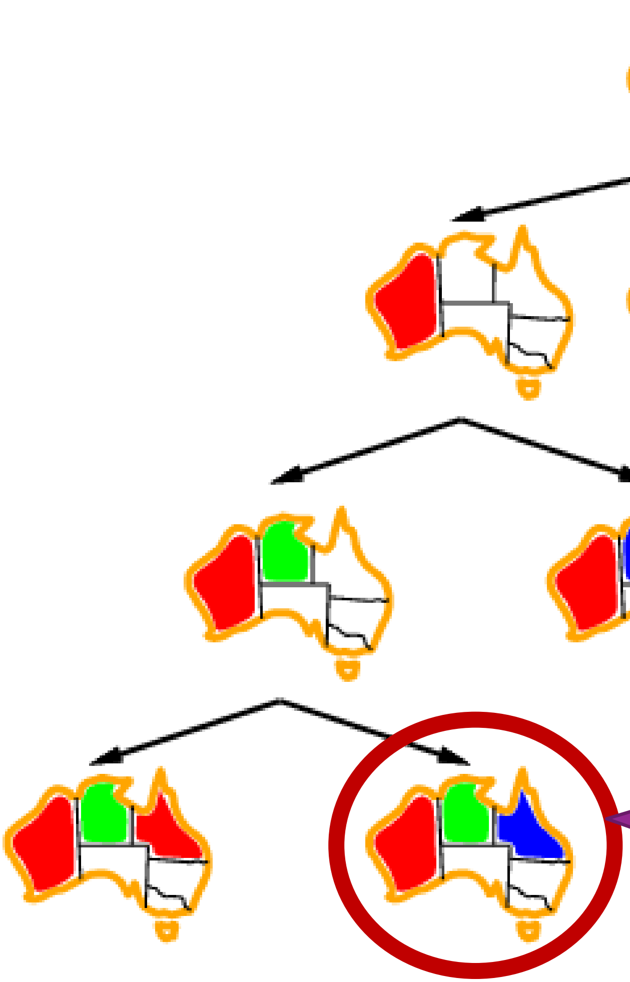
Backtracking Example



Backtracking Example



Backtracking Example



Cannot assigned SA

→ Stop

Backtracking search

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return BACKTRACK({ }, csp)
function BACKTRACK(assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    var ⟵ SELECT-UNASSIGNED-VARIABLE(csp)
    for each value in ORDER-DOMAIN-VALUES(var , assignment , csp) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ⟵ INFERENCE(csp, var , value)
            if inferences ≠ failure then
                add inferences to assignment
                result ⟵ BACKTRACK(assignment, csp)
                if result ≠ failure then
                    return result
        remove {var = value} and inferences from assignment
    return failure
```

1. Variable Selection
2. Value Selection
3. Consistency Checking

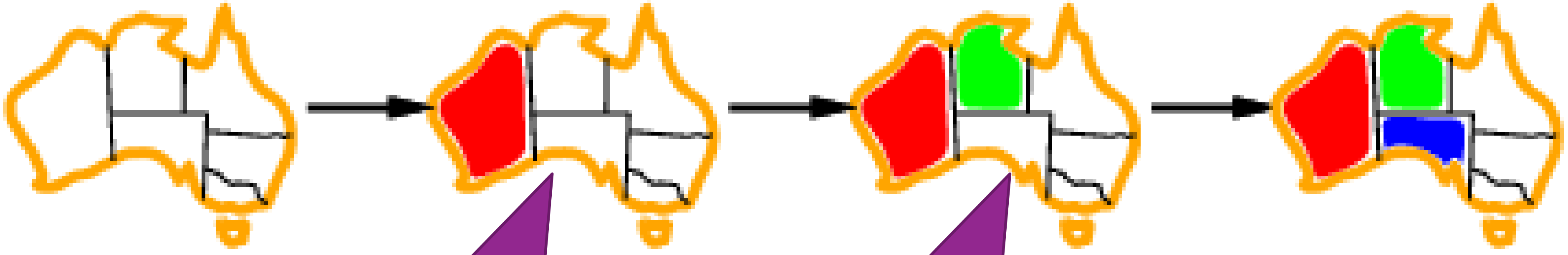
Improve Backtracking Search Efficiency

- The following general-purpose techniques can increase speed

1. Variable selection: which variable should be assigned next?
2. Value selection: In what order should its values be tried?
3. Can we detect inevitable failure early? – Use constraint propagation

Variable Selection: Minimum Remaining Values (MRV)

- **Select variable with smallest domain**

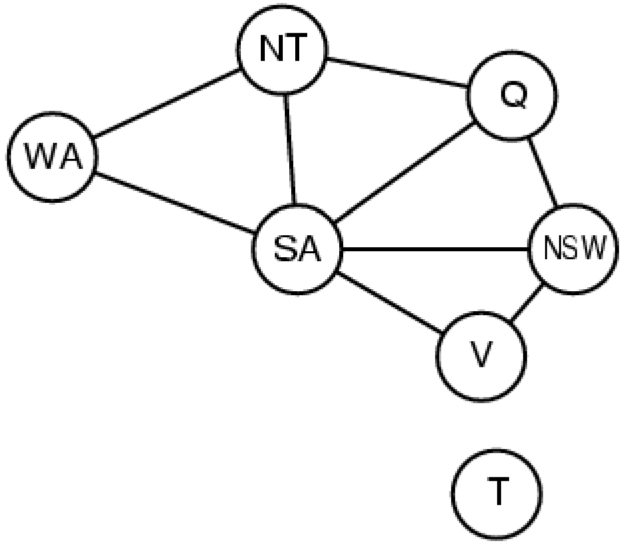


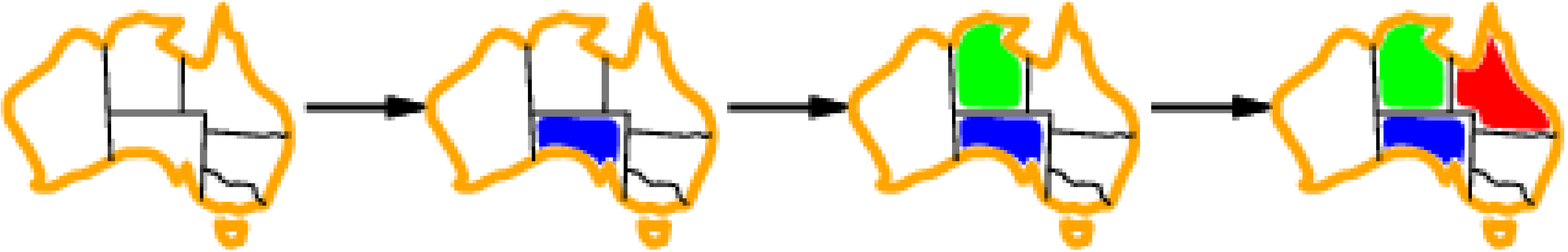
NT, SA has 2 choices

SA has only 1 choice

Variable Selection: Degree Heuristic

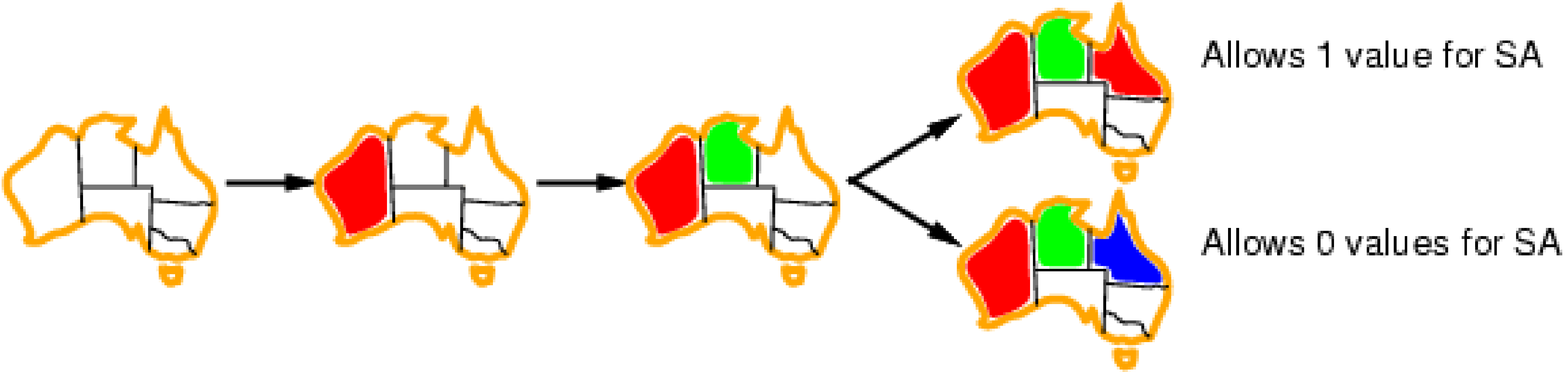
- In case of multiple MRVs
- **Choose** variable with the most constraints





Value Selection: Least Constraining Value

- Choose value that rules out the fewest values in the remaining variables



Local Search in CSPs

Local search for CSPs

- Local search typically works with “complete” states, i.e., all variables assigned
- To apply to CSPs:
  - Allow states with unsatisfied constraints
  - Actions reassign variable values
- Variable selection: randomly select any conflicted variable
- Value selection by min-conflicts heuristic:
  - choose value that violates the fewest constraints
  - example: hillclimb with $h(n)$ = total number of violated constraints

Min-Conflicts

function MIN-CONFLICTS(*csp*, *max_steps*) **returns** a solution or failure

  inputs: *csp*, a constraint satisfaction problem

    max_steps, the number of steps allowed before giving up

  current ⟵ an initial complete assignment for *csp*

  for *i* = 1 to *max_steps* **do**

    if *current* is a solution for *csp* **then return** *current*

    var ⟵ a randomly chosen conflicted variable from *csp*.VARIABLES

    value ⟵ the value *v* for *var* that minimizes CONFLICTS(*var*, *v*, *current*, *csp*)

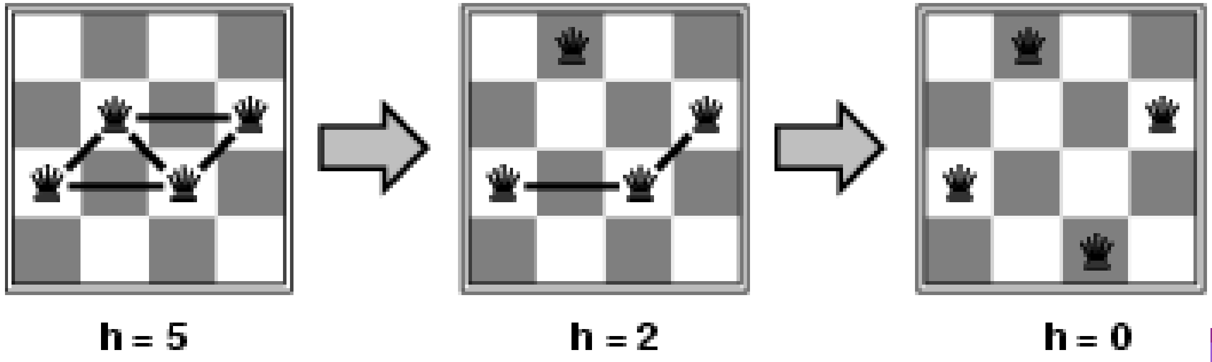
    set *var* = *value* in *current*

  return failure

ตัวอย่าง: 4-Queens

- States: 4 Queens ใน 4 Columns (4^4 = 256 States)
  - Variables: Row position of queen in each column
  - Domains: 1,2,3,4 for all
- Actions: Move queen to another row in column
- Goal test: No attacking pair
- Evaluation: $h(n)$ = number of attacking pairs

Local search can solve n-queen up to n = 10,000,000



ตัวอย่าง: 8-Queens

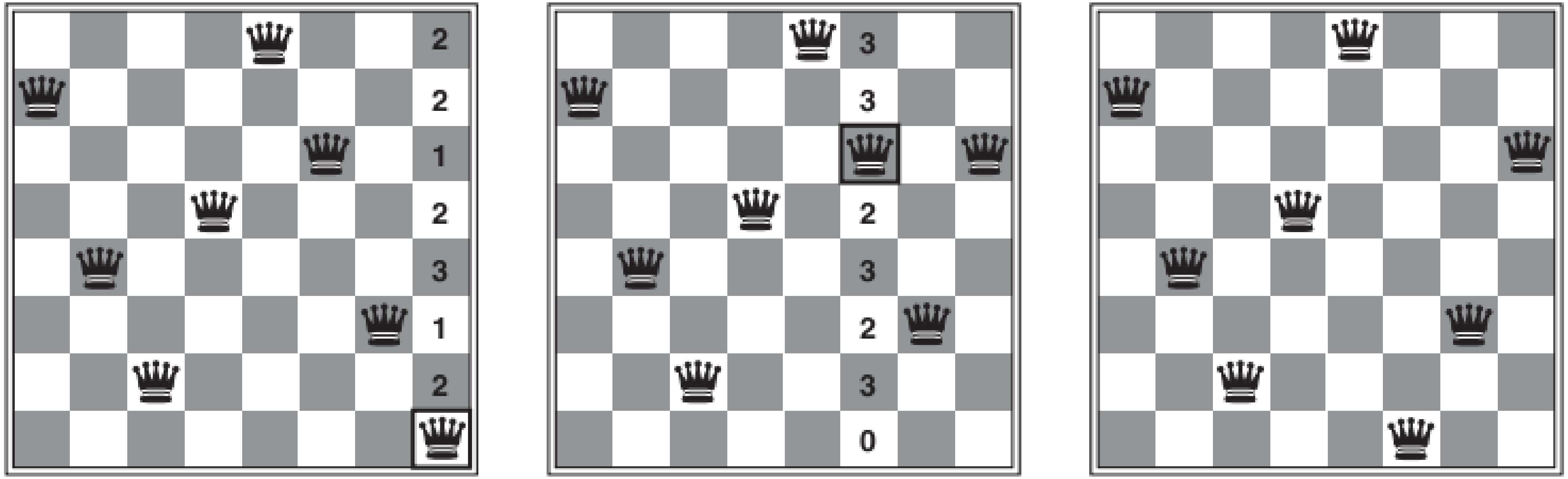


Figure 6.9 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking ties randomly.

Summary

- Local Search
  - Iterative improvement to constant number of solution in memory
  - Hillclimbing – always go to better nearby solution, stop if none exists
    - Local optima risk
    - Can use simulate annealing, tabu search to allow “going down the hill”
- CSP
  - Using constraint to reduce choice – constraint propagation
  - Tree search with backtracking
    - Order of variables and values to consider is important
  - Local search on complete (but conflicted) solution – min-conflict value reassignment

Other Issues in Search

- What is the problem is not deterministic?
  - Source: Incomplete information, unreliable actions

1. Need to manage *belief state*: set of possible states
   - Taking actions, current information can reduce number of states in belief state
2. And/or need to have *contingency plan* for each possible states
   - Search result is no longer linear path

- May not have complete understanding about the problem, or the environment is changing
  - Need to perform *online search* for one step at a time

Other Issues in Search (cont.)

- May be in *competition* environment. For example, playing chess
  - Need to take competition decision into account
  - Minimax search: assuming competition will make optimal decision i.e, worst for us