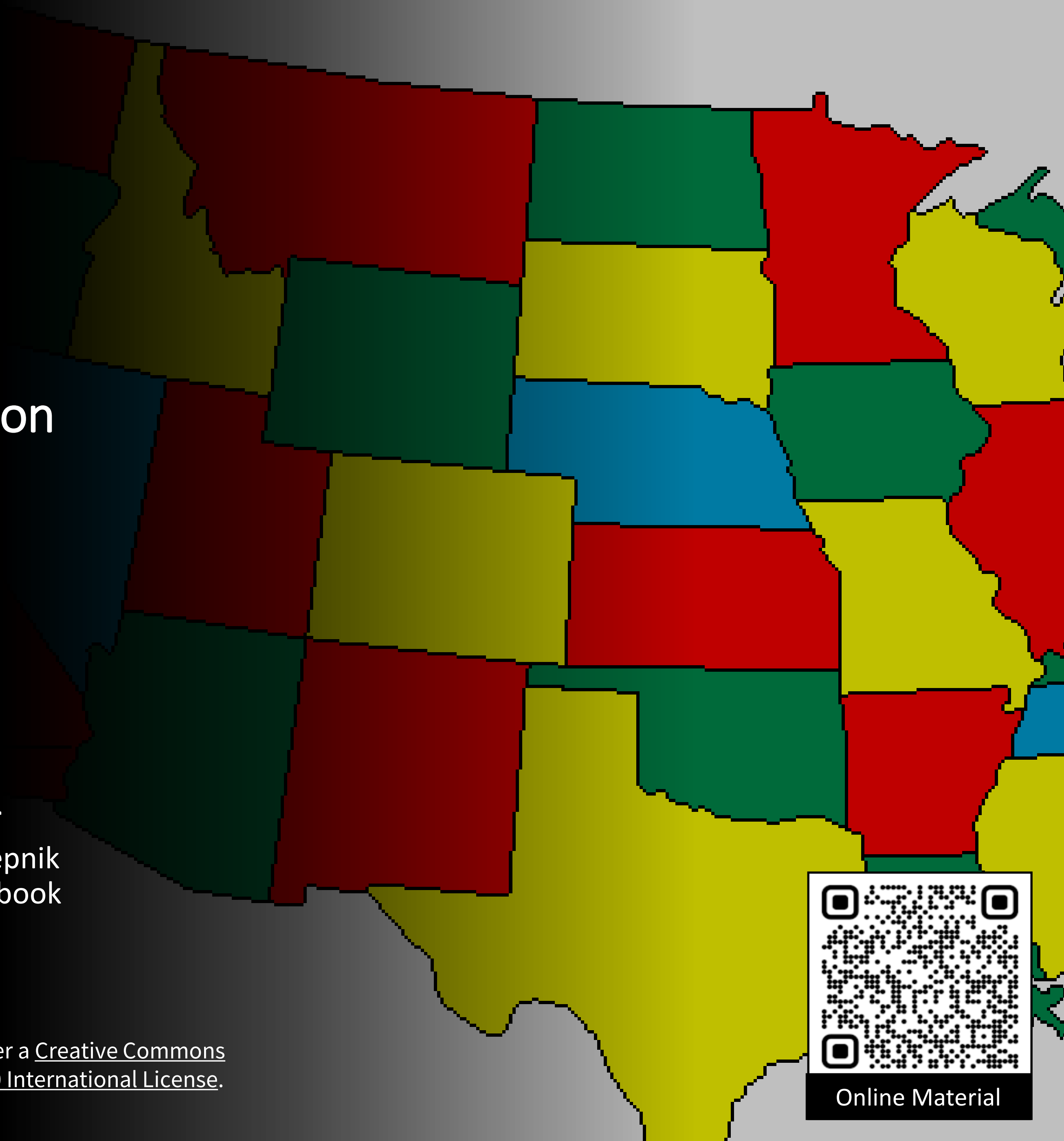


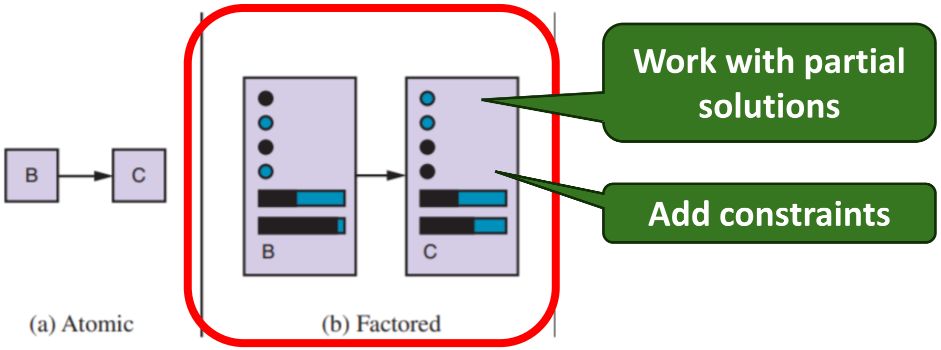
CS 5/7320
Artificial Intelligence

Constraint Satisfaction Problems
AIMA Chapter 6

Slides by Michael Hahsler
based on Slides by Svetlana Lazepnik
with figures from the AIMA textbook

Online Material

Constraint Satisfaction Problems (CSPs)



Definition:

- **State** is defined by a factored state representation:
  - A set of variables X_i called fluents.
  - Each variable can have a value from domain D_i or be **unassigned** (partial solution).

- **Constraints** are a set of rules specifying allowable combinations of values for subsets of the variables.

  E.g., $X_1 \neq X_7$ or $X_2 > X_9 + 3$

- **Solution**: a state that is a
  a) **Consistent assignment**: satisfies all constraints.
  b) **Complete assignment:** assigns value to each variable.

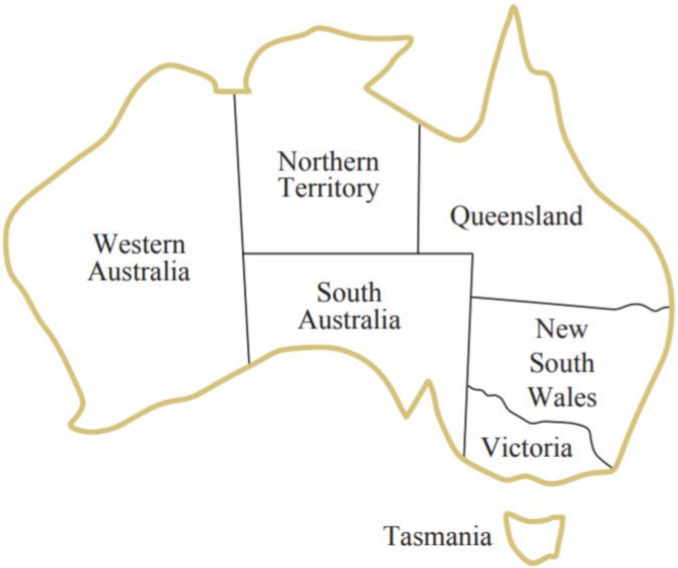
Comparison to Other Methods

	Generic Tree Search	Local Search	CSP
State representation	**Atomic** states Variables are only used to create human readable labels or calculate heuristics.	**Factored** representation to find local moves.	**Factored**
Assignment	Always **complete**	Always **complete**	**Partial** assignment during search
Constraints	Constrains are implicit in the **search problem** (initial + goal state + transition funciton).	Constraints are represented by the **objective function** and may not be met.	Enforcement of **explicit constraints**.

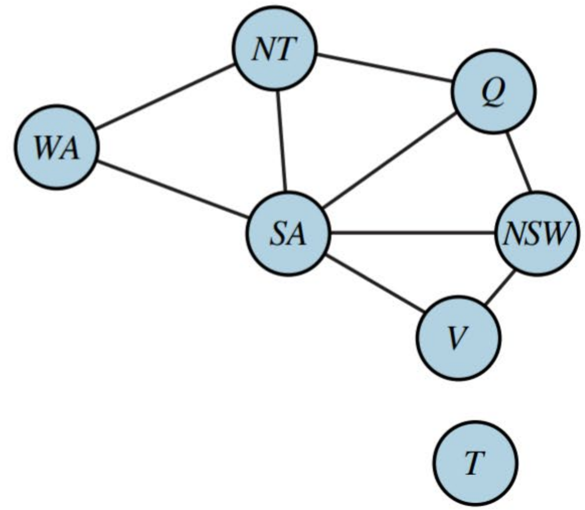
+ General-purpose solvers for CSP with more power than standard search algorithms exit.

Example: Map Coloring (Graph coloring)

Problem

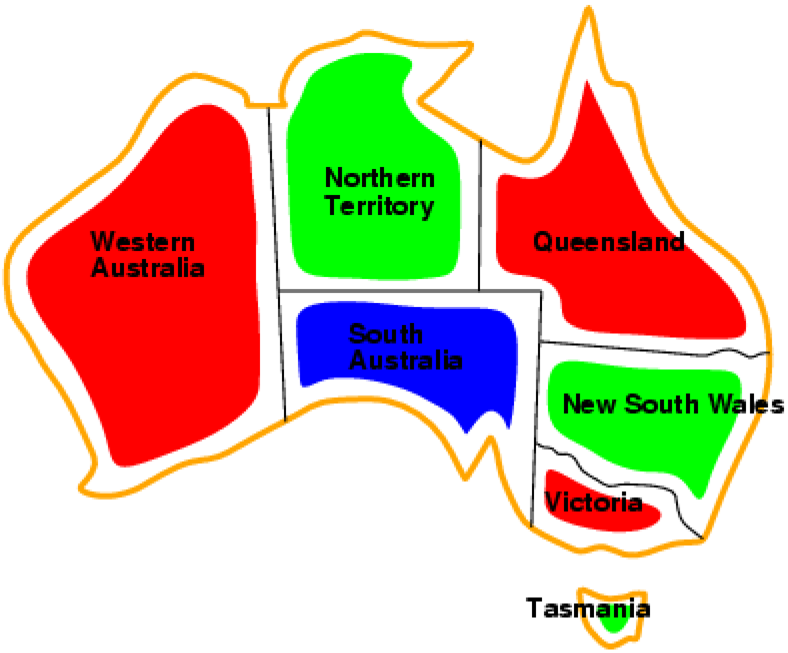


Constraint graph



- **Variables representing state:** WA, NT, Q, NSW, V, SA, T
- **Variable Domains:** {red, green, blue}
- **Constraints:** adjacent regions must have different colors e.g.,
  WA ≠ NT ⇔ (WA, NT) in {(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)}

Example: Map Coloring

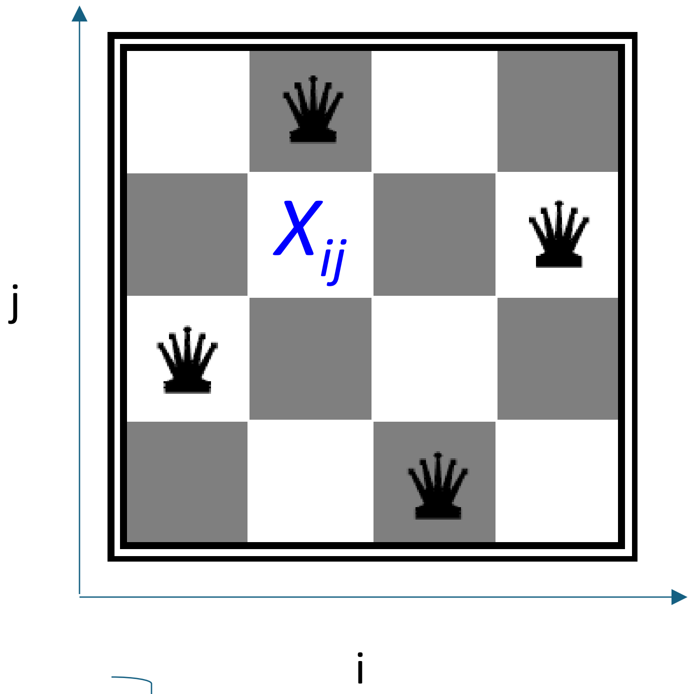


Solutions are ***complete*** and **consistent** assignments, e.g.,

WA = red, NT = green, Q = red, NSW = green,
V = red, SA = blue, T = green

Example: N-Queens

- **Variables:** X_{ij} for $i, j \in \{1, 2, \ldots, N\}$
- **Domains:** $\{0, 1\}$ # Queen: no/yes



- **Constraints:**

$\Sigma_{i,j} X_{ij} = N$

$(X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\}$ # cannot be in same col.

$(X_{ij}, X_{kj}) \in \{(0, 0), (0, 1), (1, 0)\}$ # cannot be in same row.

$(X_{ij}, X_{i+k,\, j+k}) \in \{(0, 0), (0, 1), (1, 0)\}$ # cannot be diagonal

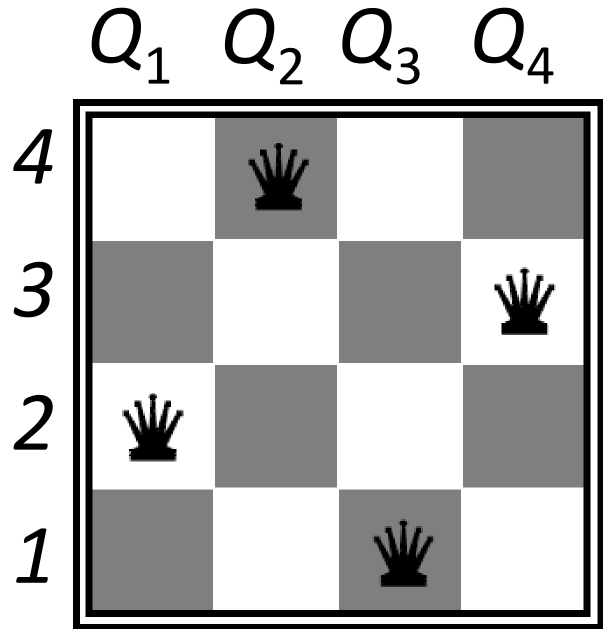
$(X_{ij}, X_{i+k,\, j-k}) \in \{(0, 0), (0, 1), (1, 0)\}$ # cannot be diagonal

for $i, j, k \in \{1, 2, \ldots, N\}$

N-Queens: Alternative Formulation

- **Variables:** $Q_1, Q_2, \dots, Q_N$
- **Domains:** $\{1, 2, \dots\ , N\}$ # row for each col.
- **Constraints:**

  $\forall$ *i*, *j* non-threatening (Q_i , Q_j)



Example:
Q1 = 2, Q2 = 4, Q3 = 1, Q4 = 3

Example: Sudoku

- **Variables:** X_{ij}
- **Domains:** {1, 2, …, 9}
- **Constraints:**

  Alldiff(X_{ij} in the same *unit)*

  Alldiff(X_{ij} in the same *row)*

  Alldiff(X_{ij} in the same *column*)

					8			4
	8	4		1	6			
			5			1		
1		3	8			9		
6		8		X_{ij}		4		3
		2			9	5		1
		7			2			
			7	8		2	6	
2			3					

Some Popular Types of CSPs

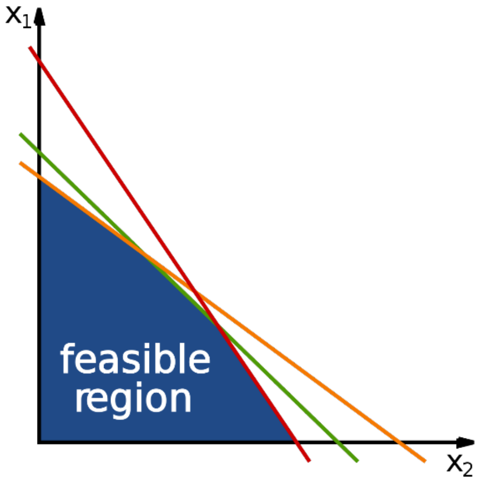
- **Boolean Satisfiability Problem (SAT)**
  Find variable assignments that makes a Boolean expression (often expressed in conjunctive normal form) evaluate as true.

$$(x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2 \lor x_3) \land \neg x_1 = \text{True}$$

- **Integer Programming**
  Variables are restricted to integers. Find a feasible solution that satisfies all constraints. The traveling salesman problem can be expressed as an integer program.

NP-complete

- **Linear Programming**
  Variables are continuous and constraints are linear (in)equalities.
  Find a feasible solution using, e.g., the simplex algorithm.



Real-world CSPs

- Assignment problems

  e.g., who teaches what class for a fixed schedule. Teacher cannot be in two classes at the same time!
- Timetable problems

  e.g., which class is offered when and where? No two classes in the same room at the same problem.
- Scheduling in transportation and production (e.g., order of production steps).
- Many problems can naturally also be formulated as CSPs.

- More examples of CSPs: http://www.csplib.org/

Formulation of a CSP as a Search Problem

State:

- Values assigned so far

Initial state:

- The empty assignment { } (all variables are unassigned)

Successor function:

- Choose an unassigned variable and assign it a value that does not violate any constraints
- Fail if no legal assignment is found

Goal state:

- Any complete and consistent assignment.

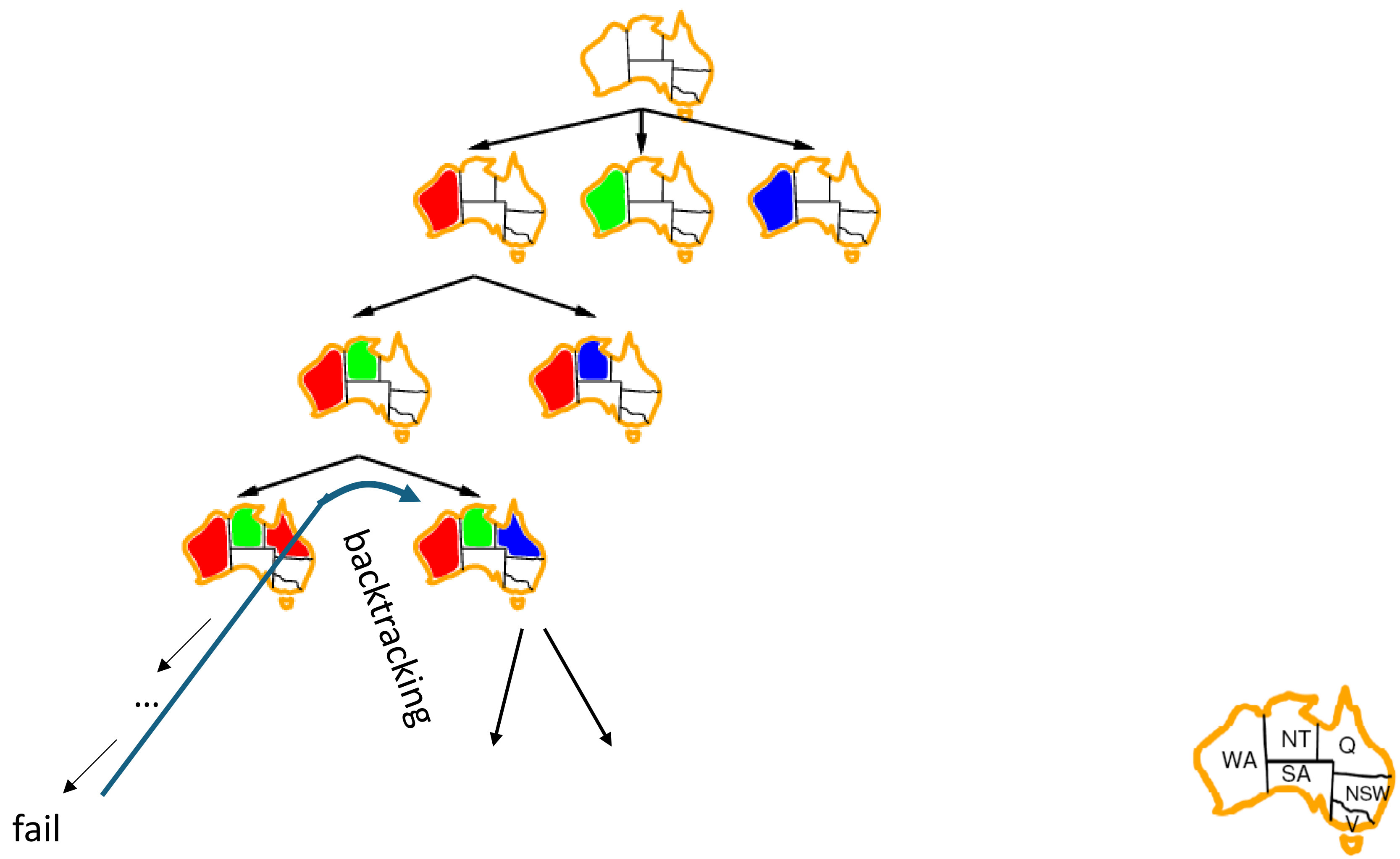
Backtracking Search

- In CSP's, variable assignments are **commutative**
  For example,
  [WA = red then NT = green] is the same as
  [NT = green then WA = red]. → Order is not important

- We can build a search tree that assigns the value to one variable per level.
  - Tree depth ***n*** (number of variables)
  - Number of leaves: d^n (d is the number of values per variable)

- Depth-first search for CSPs with single-variable assignments is called **backtracking search.**

Example: Backtracking Search (DFS)



Backtracking Search Algorithm

```
function RECURSIVE-BACKTRACKING(assignment, csp)
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp)
        if value is consistent with assignment given CONSTRAINTS[csp]
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

Call: `Recursive-Backtracking({}, csp)`

Improving backtracking efficiency:

- Which variable should be assigned next?
- In what order should its values be tried?
- Can we detect inevitable failure early?

Similar to move ordering in games.

Tree pruning (like in alpha-beta search)

Variable/Value Ordering

Which variable should be assigned next?

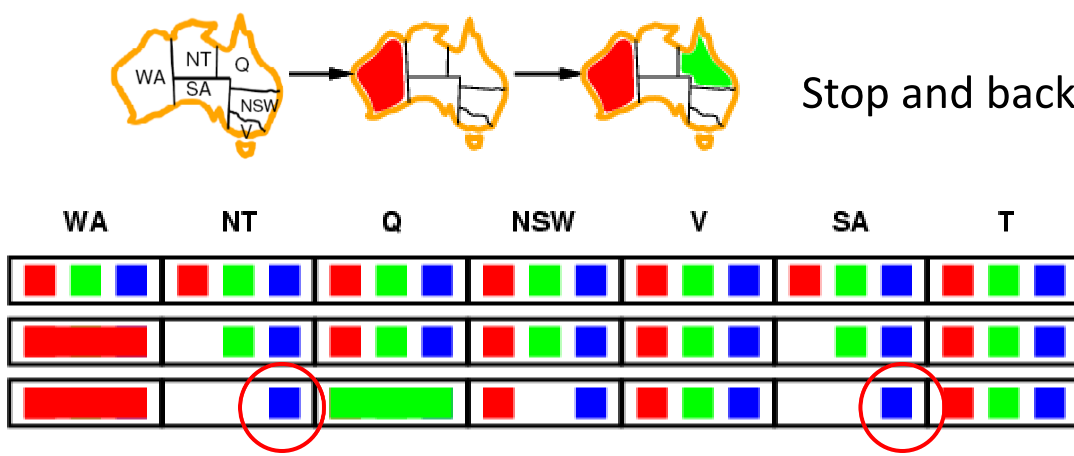
- **Most constrained variable:**
  - Keep track of remaining legal values for unassigned variables (using constraints).
  - Choose the variable with the fewest legal values left.
  - A.k.a. **minimum remaining values** (MRV) heuristic.

In which order should its values be tried?

- Choose the **least constraining value**:
  - The value that rules out the fewest values in the remaining variables.

Early Detection of Failure: Forward Checking Node Consistency

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values (i.e., minimum remaining values = 0)

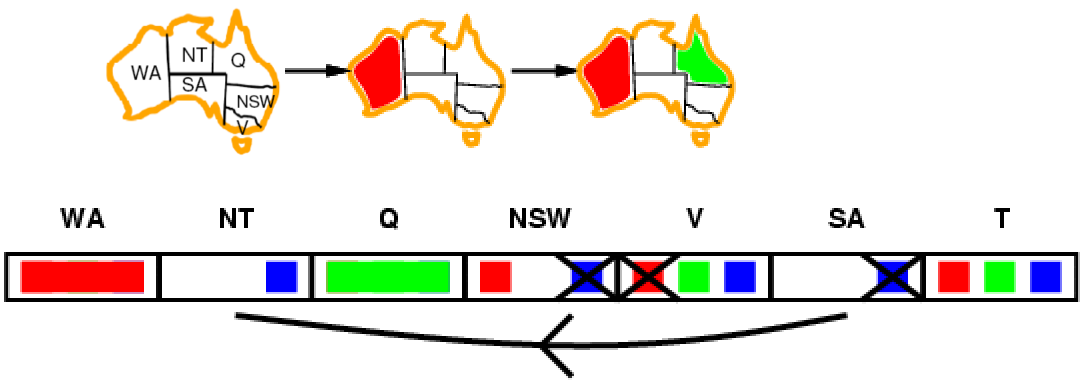


Stop and backtrack

- NT and SA cannot both be blue! This violates the constraint.

Early Detection of Failure: Forward Checking Arc Consistency

- *X* is arc consistent wrt *Y* iff for every value of *X* there is some allowed value of *Y*.
- Make *X* arc consistent wrt *Y* by throwing out any values of *X* for which there is no allowed value of *Y*.



1. NWS cannot be blue because SA has to be blue.
2. V cannot be red because NSW has to be red.
3. SA cannot be blue because NT is blue.
4. Fail and backtrack

- Arc consistency detects failure earlier than node consistency
- There are more consistency checks (path consistency, K-consistency)

Backtracking Search With Ordering and Early Failure Detection

```
function RECURSIVE-BACKTRACKING(assignment, csp)
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp)
        if value is consistent with assignment given CONSTRAINTS[csp]
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

Call: `Recursive-Backtracking({}, csp)`

```
If (inference(csp, var, assignment) == failure)
        return failure
```

Check consistency here (called "inference") and backtrack if we know that the branch will lead to failure.

Local Search for CSPs

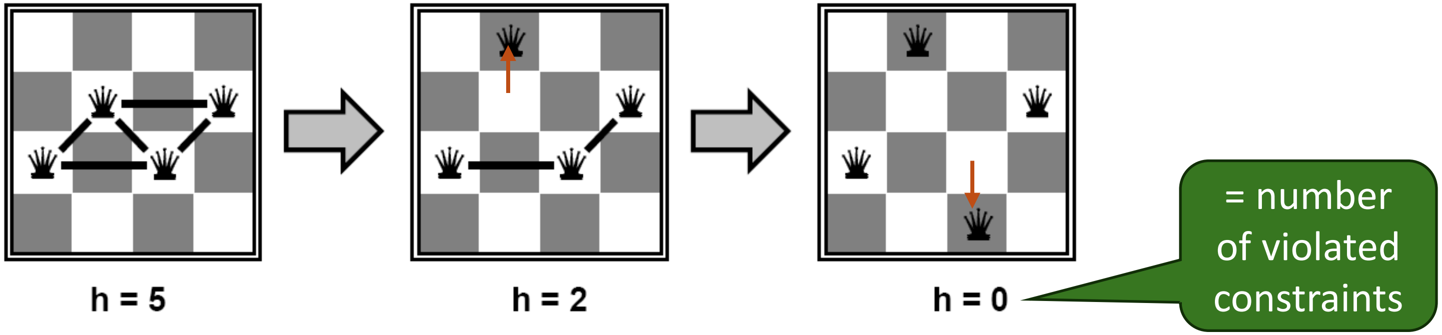
CSP algorithms
- **Allow incomplete states.**
- **States must satisfy all constraints**.

vs.

Local Search works only with
- **Only “complete” states** (all variables are assigned)
- **Allows states with unsatisfied constraints**.

Local search can attempt to reduce unsatisfied constraints by the **min-conflicts** heuristic:

1. Select a variable that violates a constraint (produces a conflict).
2. Choose a new value that violates fewer constraints.
3. Repeat till all constraints are met (or a local optimum is reached).



Local search is often very effective heuristic for CSPs.

What You Should Know

- CSPs are a special type of search problem:
  - States are **factored** and defined by a set of variables and values assignments
  - The goal is defined by a set of constraints on the variables.
  - Incomplete assignments are used to create a complete assignments piece-by-piece.
  - The goal test is defined by
    - **Consistency** with constraints
    - **Completeness** of assignment

- Many problems can be formulated as a CSP and problems where the constraints are very restrictive on the solution space may be easier to solve as CSPs (e.g., scheduling problems and puzzles).

- Effective off-the-shelf solvers exist. They typically use:
  - **Depth-first search**: successor states are generated variable-by-variable by adding a consistent value assignment to single unassigned variables.
  - **Local search** can be used as an effective heuristic. It search the space of all complete assignments for consistent assignments = **min-conflicts heuristic.**