Artificial Intelligence Foundation – JC3001

Lecture 14: Constraint Satisfaction Problems II

Prof. Aladdin Ayesh (aladdin.ayesh@abdn.ac.uk)

Dr. Binod Bhattarai (binod.bhattarai@abdn.ac.uk)

Dr. Gideon Ogunniye, (g.ogunniye@abdn.ac.uk)

September 2025

Material adapted from:
Russell and Norvig (AIMA Book): Chapter 6

Course Progression

- Part 1: Introduction
  1. Introduction to AI ✓
  2. Agents ✓
- Part 2: Problem-solving
  1. Search 1: Uninformed Search ✓
  2. Search 2: Heuristic Search ✓
  3. Search 3: Local Search ✓
  4. Search 4: Adversarial Search ✓
- Part 3: Reasoning and Uncertainty
  1. **Reasoning 1: Constraint Satisfaction**
  2. Reasoning 2: Logic and Inference
  3. Probabilistic Reasoning 1: BNs
  4. Probabilistic Reasoning 2: HMMs
- Part 4: Planning
  1. Planning 1: Intro and Formalism
  2. Planning 2: Algos and Heuristics
  3. Planning 3: Hierarchical Planning
  4. Planning 4: Stochastic Planning
- Part 5: Learning
  1. Learning 1: Intro to ML
  2. Learning 2: Regression
  3. Learning 3: Neural Networks
  4. Learning 4: Reinforcement Learning
- Part 6: Conclusion
  1. Ethical Issues in AI
  2. Conclusions and Discussion

Objectives

- Defining Constraint Satisfaction Problems (CSP) ✓
- CSP examples ✓
- Backtracking search for CSPs
- Local search for CSPs
- Problem structure and problem decomposition

- Variable assignments are commutative, i.e.,
  $[WA = red$ **then** $NT = green]$ same as $[NT = green$ **then** $WA = red]$
- Only need to consider assignments to a single variable at each node
  $\Rightarrow b = d$ 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 for $n \approx 25$

Backtracking search algorithm

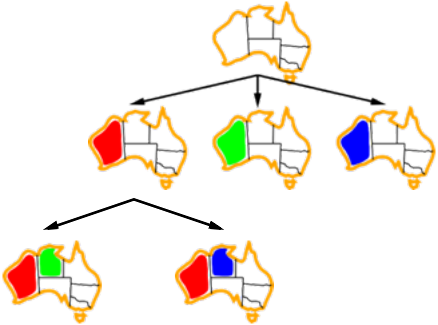
function Backtracking-Search(csp) **returns** a solution or $failure$
  return Backtrack($\{\}$, csp)

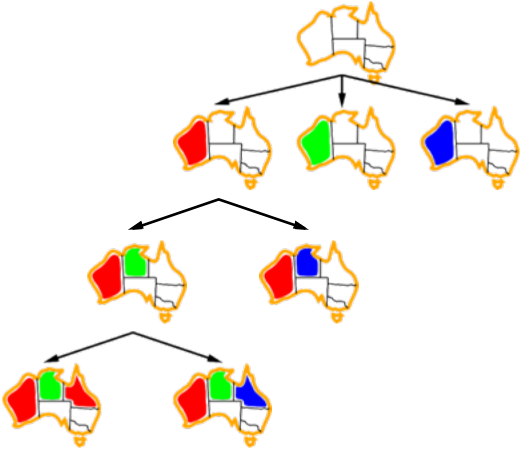
---

function Backtrack(csp, $assignment$) **returns** a solution or $failure$
  if $assignment$ is complete **then return** $assignment$
  $var \leftarrow$ Select-Unassigned-Variable$(csp, assignment)$
  for each $value$ in Order-Domain-Values$(csp, var, assignment)$ **do**
    if $value$ is consistent with $assignment$ **then**
      add $\{var = value\}$ to $assignment$
      $inferences \leftarrow$ Inference$(csp, var, assignment)$
      if $inferences \neq failure$ **then**
        add $inferences$ to csp
        $result \leftarrow$ Backtrack$(csp, assignment)$
        if $result \neq failure$ **then return** $result$
        remove $inferences$ from csp
      remove $\{var = value\}$ from $assignment$
  return $failure$

Backtracking example

Backtracking example

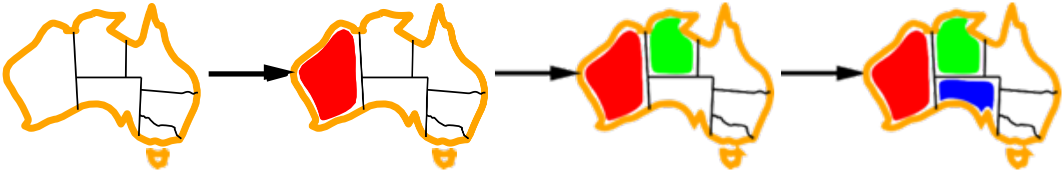




General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next (Select-Unnassigned-Variable)?
2. In what order should we try its values (Order-Domain-Values)?
3. Can we detect inevitable failure early (Inference and Backtrack)?
4. Can we take advantage of problem structure?
5. Can we save and reuse partial results from the search?

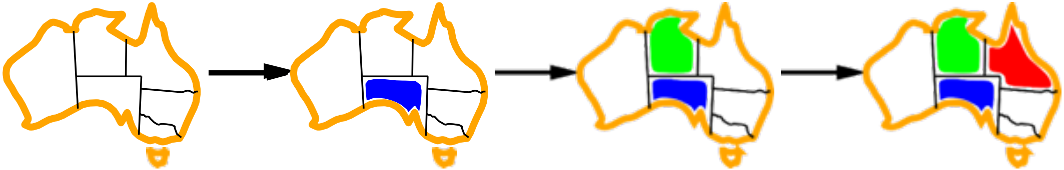
Minimum remaining values (MRV): choose the variable with the fewest legal values



Tie-breaker among MRV variables

Degree heuristic:

- choose the variable with the most constraints on remaining variables

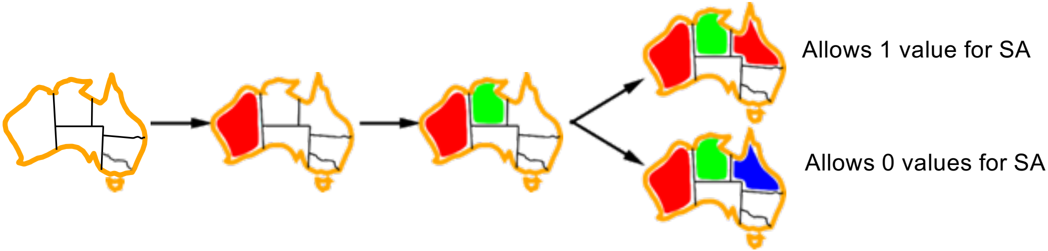


Rationale:

- Most constrained variables will likely fail first, avoiding fruitless search

Given a variable, choose the least constraining value:

- the one that rules out the fewest values in the remaining variables



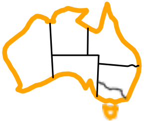
Rationale:

- Combining these heuristics makes 1000 queens feasible

Forward checking

Idea: Keep track of remaining legal values for unassigned variables

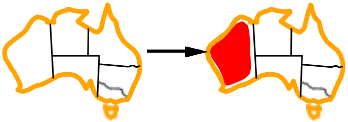
Terminate search when any variable has no legal values

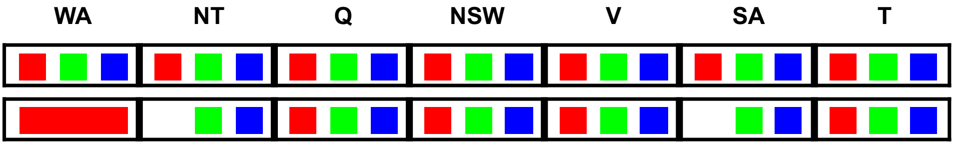


Forward checking

Idea: Keep track of remaining legal values for unassigned variables

Terminate search when any variable has no legal values

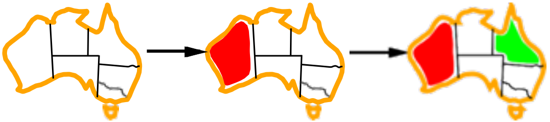


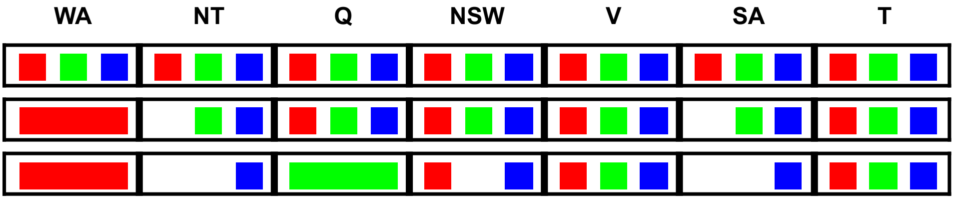


Forward checking

Idea: Keep track of remaining legal values for unassigned variables

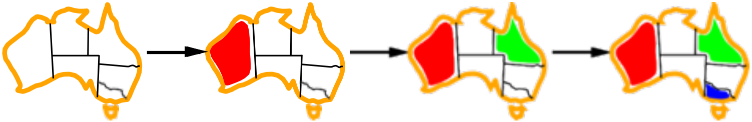
Terminate search when any variable has no legal values

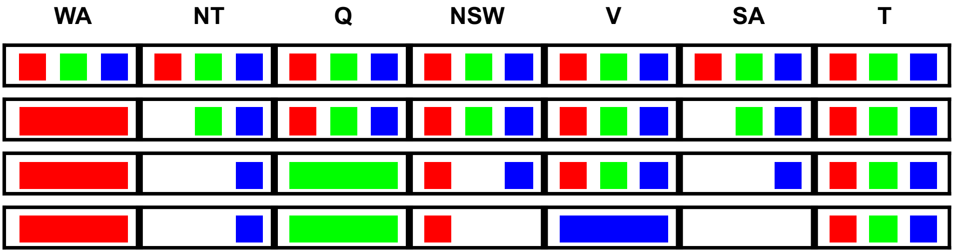




Forward checking

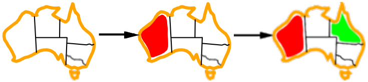
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values





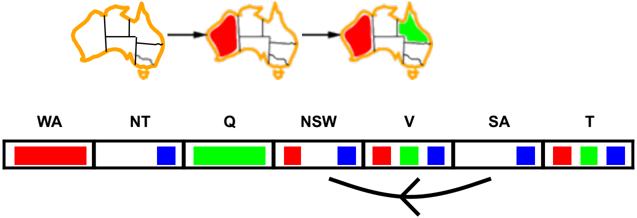
Constraint propagation

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

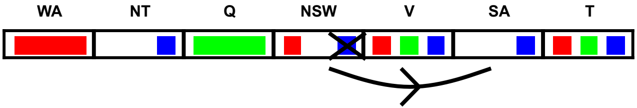


- NT and SA cannot both be blue!
- **Constraint propagation** enforces constraints locally

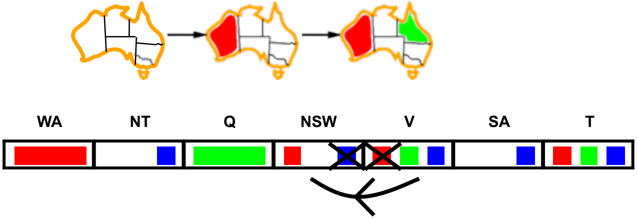
- The simplest form of propagation makes each arc **consistent**
- $X \to Y$ is consistent iff
  for **every** value x of X there is **some** allowed y



- The simplest form of propagation makes each arc **consistent**
- $X \to Y$ is consistent iff
  for **every** value x of X there is **some** allowed y

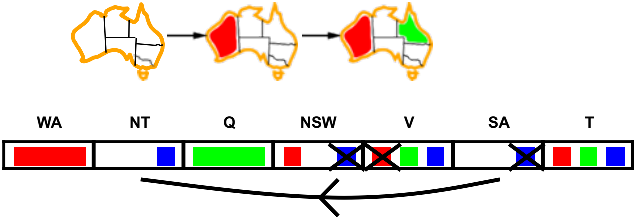


- The simplest form of propagation makes each arc **consistent**
- $X \to Y$ is consistent iff
  for **every** value x of X there is **some** allowed y



- If X loses a value, neighbours of X need to be rechecked

- The simplest form of propagation makes each arc **consistent**
- $X \to Y$ is consistent iff
  for **every** value x of X there is **some** allowed y



- If X loses a value, neighbours of X need to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor, or after each assignment

Arc consistency algorithm

function AC-3(csp) **returns** the csp, possibly with reduced domains
  $queue \leftarrow$ a queue of arcs, initially all the arcs in csp
  while $queue$ is not empty **do**
    $(X_i, X_j) \leftarrow$ POP$(queue)$
    if Revise(csp, X_i, X_j) **then**
      if size of $D_i = 0$ **then return** $false$
      for each X_k in $X_i.Neighbors - \{X_j\}$ **do**
        add (X_k, X_i) to $queue$
  return $true$

---

function Revise(csp, X_i, X_j) **returns** true iff we revise the domain of X_i
  $revised \leftarrow false$
  for each $x \in D_i$ **do**
    if no value y in D_j allows (x, y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      $revised \leftarrow true$

$O(n^2d^3)$ can be reduced to $O(n^2d^2)$ (but detecting **all** is NP-hard)

To continue in the next session.