Constraint Satisfaction Problems

Lecture 9: Artificial Intelligence

Dr. Bonaventure Chidube Molokwu

bonaventure.molokwu[at]csus.edu

College of Engineering and Computer Science
California State University
Sacramento
California - United States of America

Constraint Satisfaction Problems

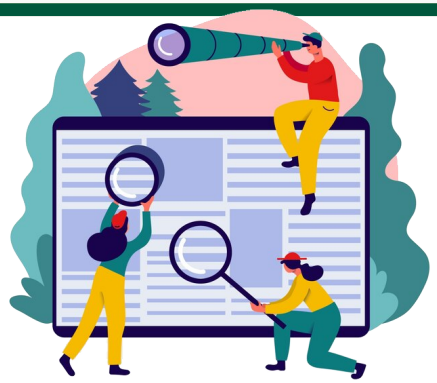
Lecture/Week Outline & Learning Outcomes

1. **Lesson/Week Outline:**
   - 1.1 Constraint Satisfaction Problem (CSP)

2. **Learning Outcomes:**
   - 2.1 Defining the concept of Constraint Satisfaction Problem (CSP).
   - 2.2 Examining examples of CSPs.
   - 2.3 Backtracking search for CSPs.
   - 2.4 Local search for CSPs.
   - 2.5 Problem structure and problem decomposition.

Constraint Satisfaction Problems
Prelude

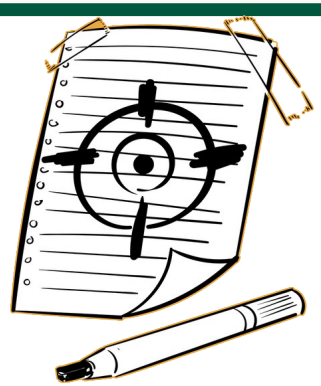
Constraint Satisfaction Problems
Introduction



- **formulation(Problem):**
  - agent(Problem-Solving): an agent which plans ahead to find a sequence of actions that will eventually achieve its goal.
  - State: a situation that an agent(Problem-Solving) can find itself in.
  - state(Initial): start point of an agent(Problem-Solving).
  - state(Goal): end point of an agent(Problem-Solving).
  - function(Successor): set of action-state-cost triples. It instructs the algorithm(Search) on what moves are possible from the state(Current), what state(Result) can be reached from each given state, and what it costs per action wrt. reaching a state(Result).

$$\text{successor}(s) = \{(a_1, s_1, c_1), (a_2, s_2, c_2), \ldots\} \text{ where } s = \text{current state};$$
$$a_i = \text{action taken}; \quad s_i = \text{resultant state after using action, } a_i;$$
$$c_i = \text{cost of moving from } s \text{ to } s_i.$$

Constraint Satisfaction Problems
Introduction

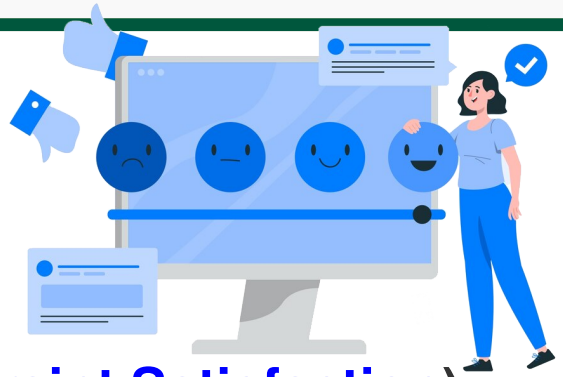


- **formulation(Problem):**
  - Path: a sequence/collection of actions forms a path.
  - Solution: a path from the state(Initial) to the state(Goal).
  - solution(Optimal): a solution whose path, from the state(Initial) to the state(Goal), has the *lowest* cost.

- **problem(Constraint Satisfaction):**

  A CSP is comprised of three (3) main components, viz:
  - $X =$ a set of variables, $\{X_1, \ldots, X_n\}$.
  - $D =$ a set of domains, $\{D_1, \ldots, D_n\}$ such that $\forall\ D_i \rightarrow X_i$.
  - $C =$ a set of constraints that specify allowable combination of values from a domain, D, to a variable, X.

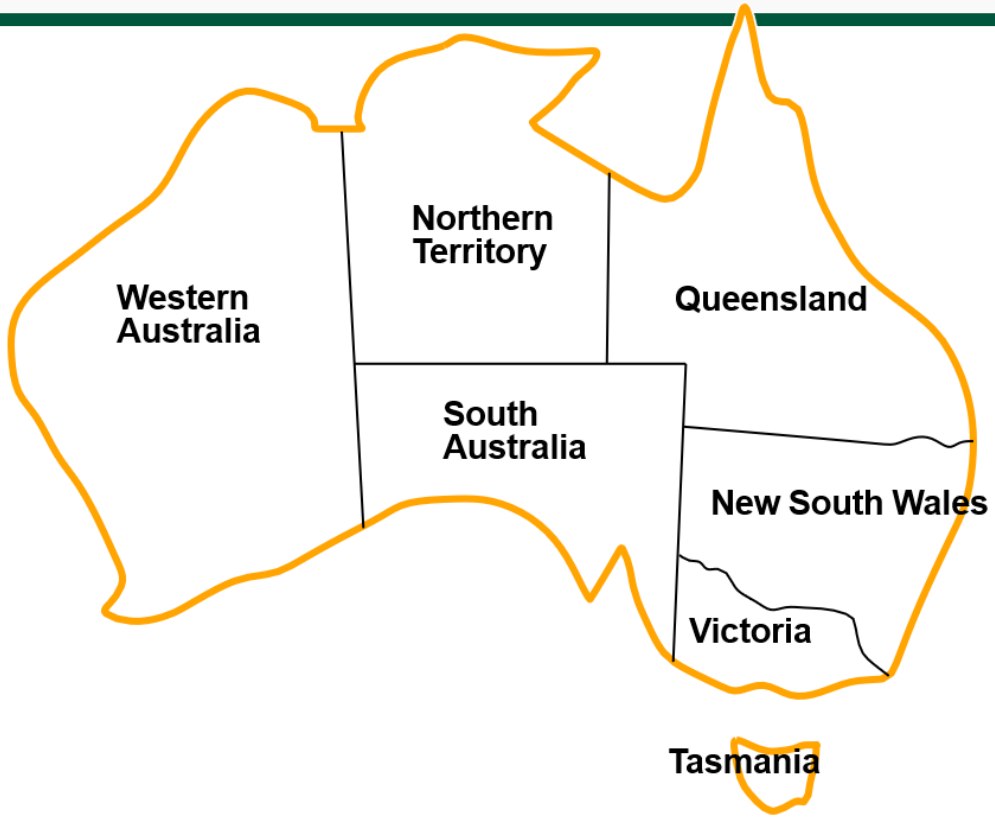
Constraint Satisfaction Problems
Introduction



- **problem(Constraint Satisfaction):**
  - Each domain, D_i, is comprised of a set of permissible values, $\{v_1, \ldots, v_k\}$, which are mapped to a variable, X_i.

    Thus, a CSP deals with the assignments of values, $v_m \in D_i$, to variables, X_i, such that $\{X_i = v_m \in D_i, \quad X_j = v_m \in D_j, \quad \ldots\}$
  - assignment(Consistent/Legal): An *assignment*, $v_m \in D_i \rightarrow X_i$, that does NOT violate any predefined constraints.
  - assignment(Complete): An *assignment*, $v_m \in D_i \rightarrow X_i$, where every variable, X_i, is mapped to a value, $v_m \in D_i$, from a domain.
  - assignment(Partial): An *assignment*, $v_m \in D_i \rightarrow X_i$, where *some* variables, X_i, are NOT mapped to a value, $v_m \in D_i$, from a domain.
  - solution(Complete): assignment(Complete) + assignment(Consistent/Legal). Thus, a solution to a CSP must be a solution(Complete).
  - solution(Partial): assignment(Partial) + assignment(Consistent/Legal).

Constraint Satisfaction Problems
Formulation of a CSP



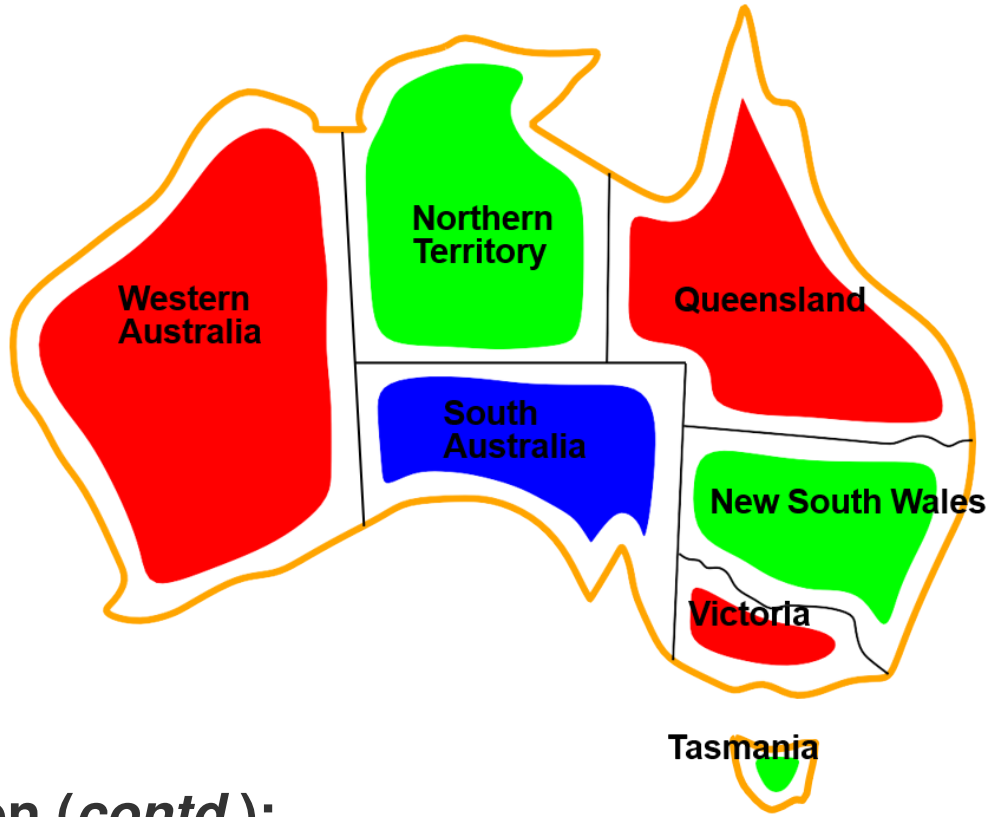
- **Example 1 - problem(Constraint Satisfaction):**
  Coloring Problem: Color ∀ region of the map above with either Red, Green, or Blue such that NO two (2) neighboring regions have the same color.

  Solution:
  - Firstly, we define the variables → regions:
    $X = \{WA, NT, SA, Q, NSW, V, T\}$
  - Secondly, the domain of every variable is the set, $D_i = \{red, green, blue\}$.

Constraint Satisfaction Problems
Formulation of a CSP

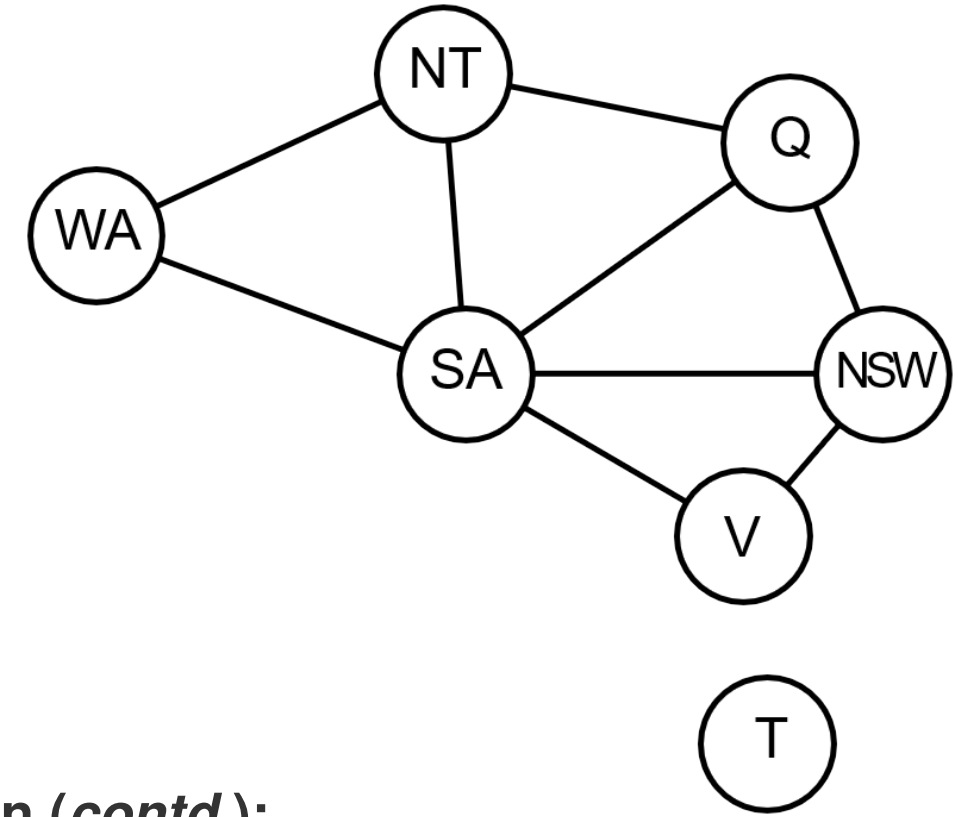
- **Solution (*contd.*):**
  - Thirdly, the constraints require regions(Neighbor) to have distinct colors. Thus, since there are nine (9) places where regions border, then there should be nine (9) constraints, viz:
    $C = \{WA \neq NT,\ WA \neq SA,\ NT \neq SA,\ NT \neq Q,\ SA \neq Q,\ SA \neq NSW,\ SA \neq V,\ Q \neq NSW,\ NSW \neq V\}$
  - Possible pairs of colors wrt. domain, $D_i =$ $\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}$

Constraint Satisfaction Problems
Formulation of a CSP



- **Solution (*contd.*):**
  - There are many possible solutions to this CSP problem.
  - One (1) possible solution: $\{WA = red,\ NT = green,\ Q = red,\ NSW = green,\ V = red,\ SA = blue,\ T = red\}$

Constraint Satisfaction Problems
Formulation of a CSP



- **Solution (*contd.*):**
  - It's helpful to visualize a CSP as a graph(Constraint) as shown above.
  - graph(Constraint): a graph where its nodes/vertices correspond to variables_X(CSP), and every edge/link of the graph connects any two (2) variables_X(X_1, X_2) that participate in a constraint.

Constraint Satisfaction Problems
Formulation of a CSP

- **Example 2 - problem(Constraint Satisfaction):**
  Job-Shop Scheduling: Consider a car-assembly scheduling problem. The whole job is composed of tasks, and we can model each task as a variable, X_i; where the value, v_m, assigned to each variable, X_i, is the time (no. of mins.). A constraint asserts that one task must occur before another begins.

  Solution:
  - Firstly, we define the variables, X_i, $\rightarrow$ tasks herein:
    $$X = \{AxleF, AxleB, WheelLF, WheelRF, WheelLB, WheelRB\}$$
  - Secondly, the domain of every variable, X_i, is the set, $D_i = \mathbb{Z}^+$.
  - Thirdly, if a task starts at time, T_1, and takes duration, d_1, to complete AND it must occur before another task starts at time, T_2; then we define it as an arithmetic constraint of the form: $(T_1 + d_1) \leq T_2$

Constraint Satisfaction Problems
Formulation of a CSP

- **Solution (*contd.*):**

  • Thus, if the *axles* have to be in place before the *wheels* are put on; and it takes 10 mins. to install an *axle*, so we write our constraints as, viz:

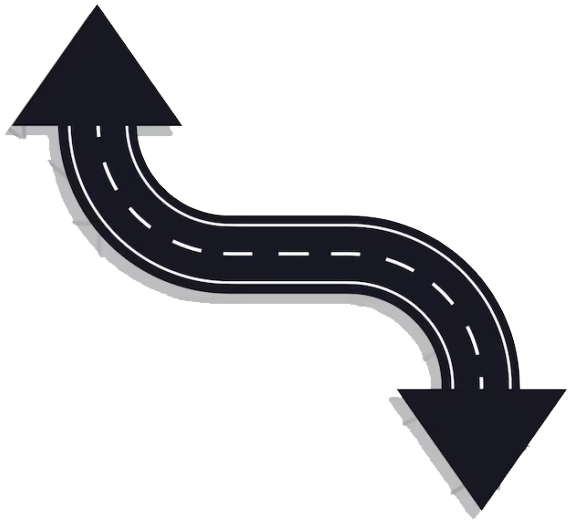
$$(T_{AxleF} + 10) \leq T_{WheelLF}$$

$$(T_{AxleF} + 10) \leq T_{WheelRF}$$

$$(T_{AxleB} + 10) \leq T_{WheelLB}$$

$$(T_{AxleB} + 10) \leq T_{WheelRB}$$

Constraint Satisfaction Problems
Backtracking Search for CSPs



- **Backtracking Search:**
  - CSP search algorithms (e.g Backtracking Search) use heuristics(General) or heuristics(Domain-Independent) rather than heuristics(Domain-Specific) to facilitate the solution of CSPs.
  - Main idea is to eliminate large portions of the space(Search) all at once by identifying variable/value combinations that violate the constraints.
  - search(Backtracking) only needs to consider *assignments* to a single variable at each node.
- search(Backtracking) == search(DepthFirst) with single-variable assignment at each node.

Constraint Satisfaction Problems

Backtracking Search for CSPs

Algorithm 1 Backtracking-Search algorithm

```
1:  function BACKTRACKSEARCH(csp, assignments)
2:      if assignments == 'complete' then
3:          return assignments
4:      end if

5:      uaVar = selectUnassignedVariable(csp, assignments)
6:      for each value in OrderDomainValues(uaVar, csp, assignments) do
7:          if isConsistent(value, uaVar) then
8:              add(value → uaVar) onto assignments
9:              inferences = makeInference(uaVar, csp, assignments)

10:             if inferences ≠ 'failure' then
11:                 add(inferences) onto csp
12:                 result = BackTrackSearch(csp, assignments)
13:                 if result ≠ 'failure' then
14:                     return result
15:                 end if
16:             end if
17:         end if
18:     end for

19:     return 'failure'
20: end function
```

Constraint Satisfaction Problems
Backtracking Search for CSPs: Coloring Problem

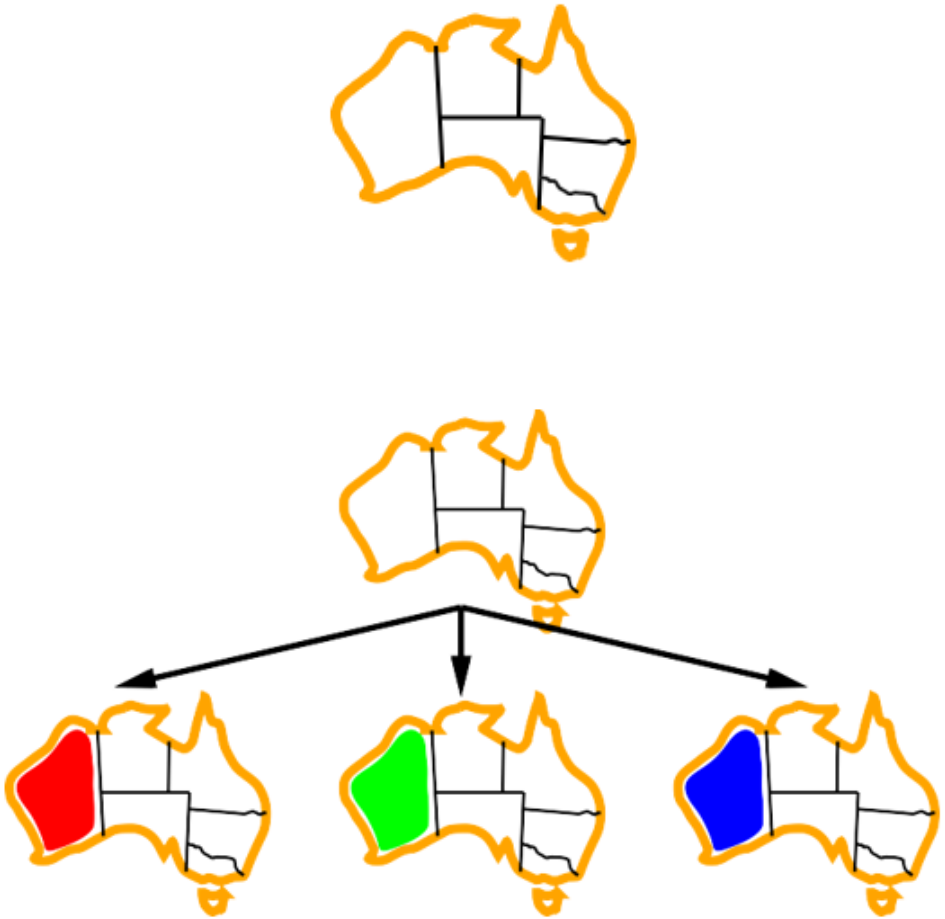


Figure: Trees(1..2) wrt. search(Backtracking) for `Coloring Problem`

- **search(Backtracking):**
  - Can be depicted as a search(Tree) with single-variable assignment at each node.

Constraint Satisfaction Problems
Backtracking Search for CSPs: Coloring Problem

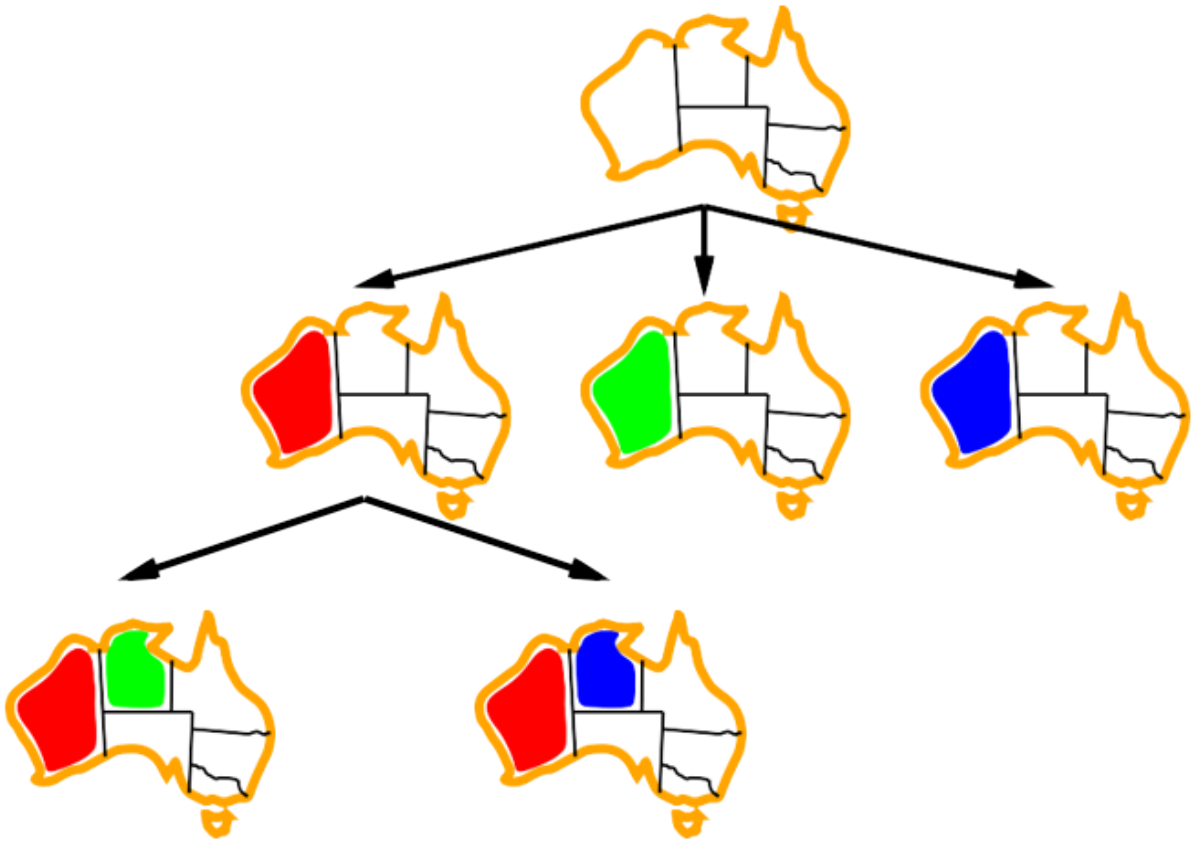


Figure: Trees(3) wrt. search(Backtracking) for `Coloring Problem`

- **search(Backtracking):**
  - Can be depicted as a search(Tree) with single-variable assignment at each node.

Constraint Satisfaction Problems

Backtracking Search for CSPs: Coloring Problem

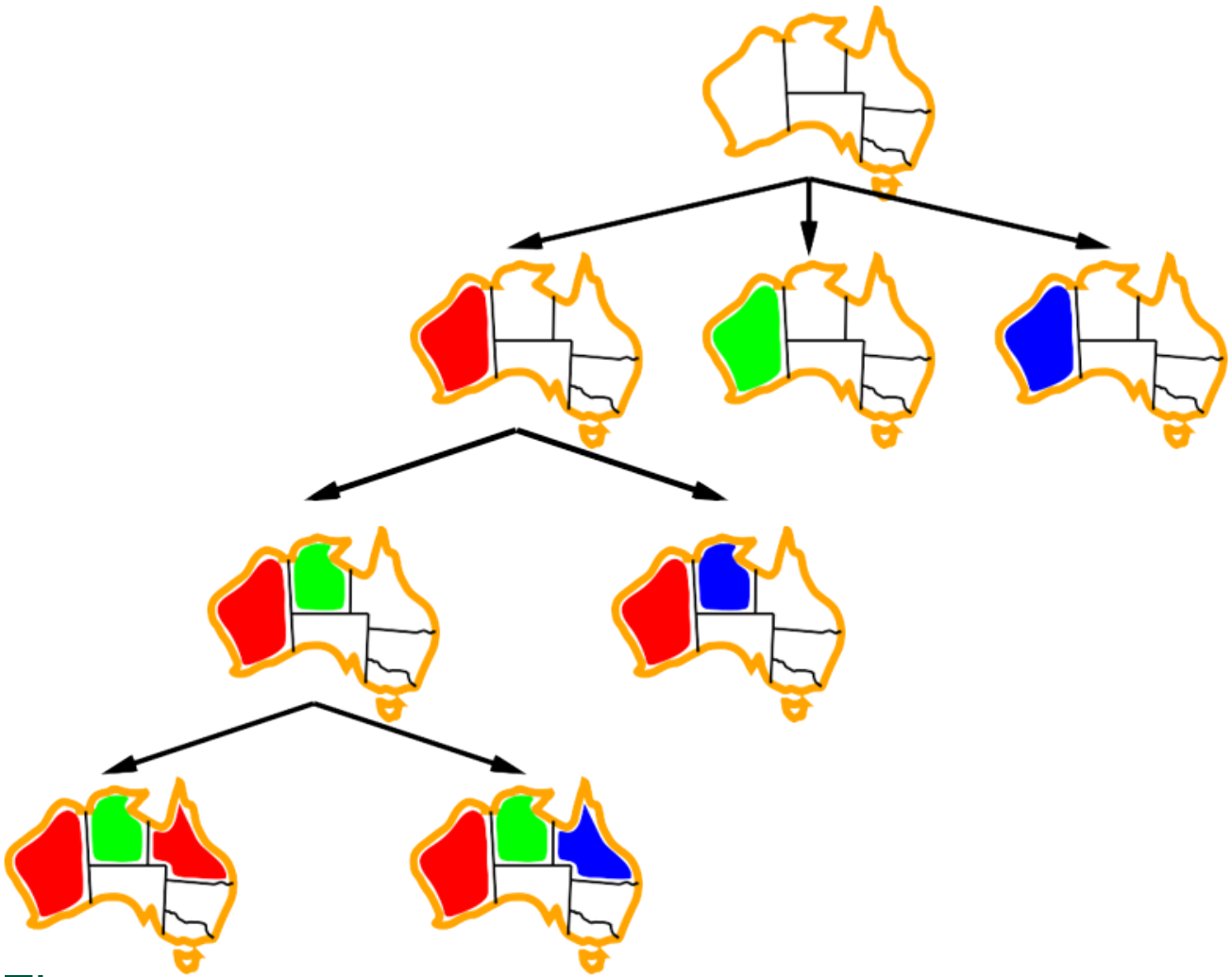


Figure: Trees(4) wrt. search(Backtracking) for `Coloring Problem`

Constraint Satisfaction Problems

Backtracking Search for CSPs: Coloring Problem

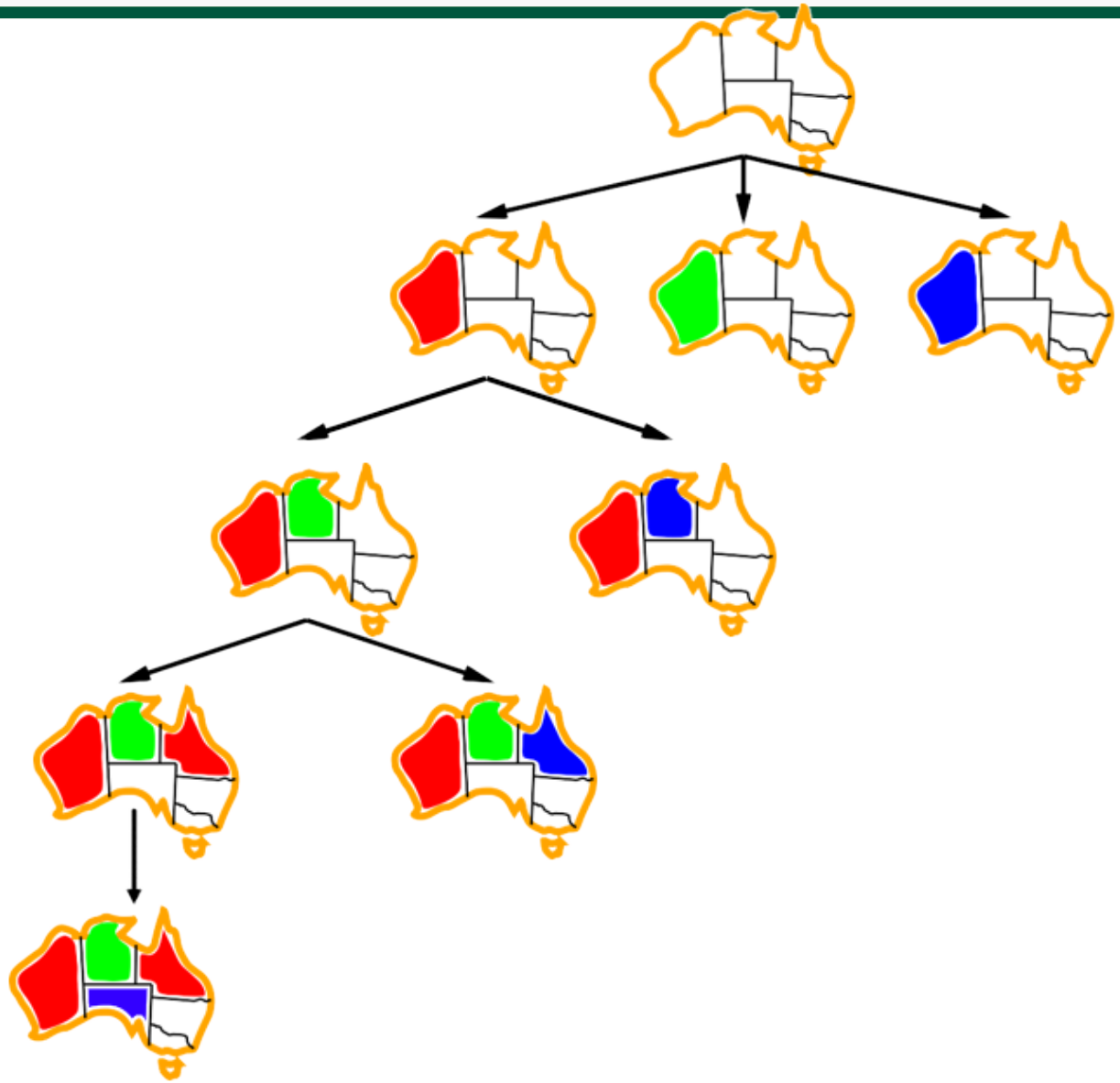


Figure: Trees(5) wrt. search(Backtracking) for `Coloring Problem`

Constraint Satisfaction Problems

Backtracking Search for CSPs: Coloring Problem

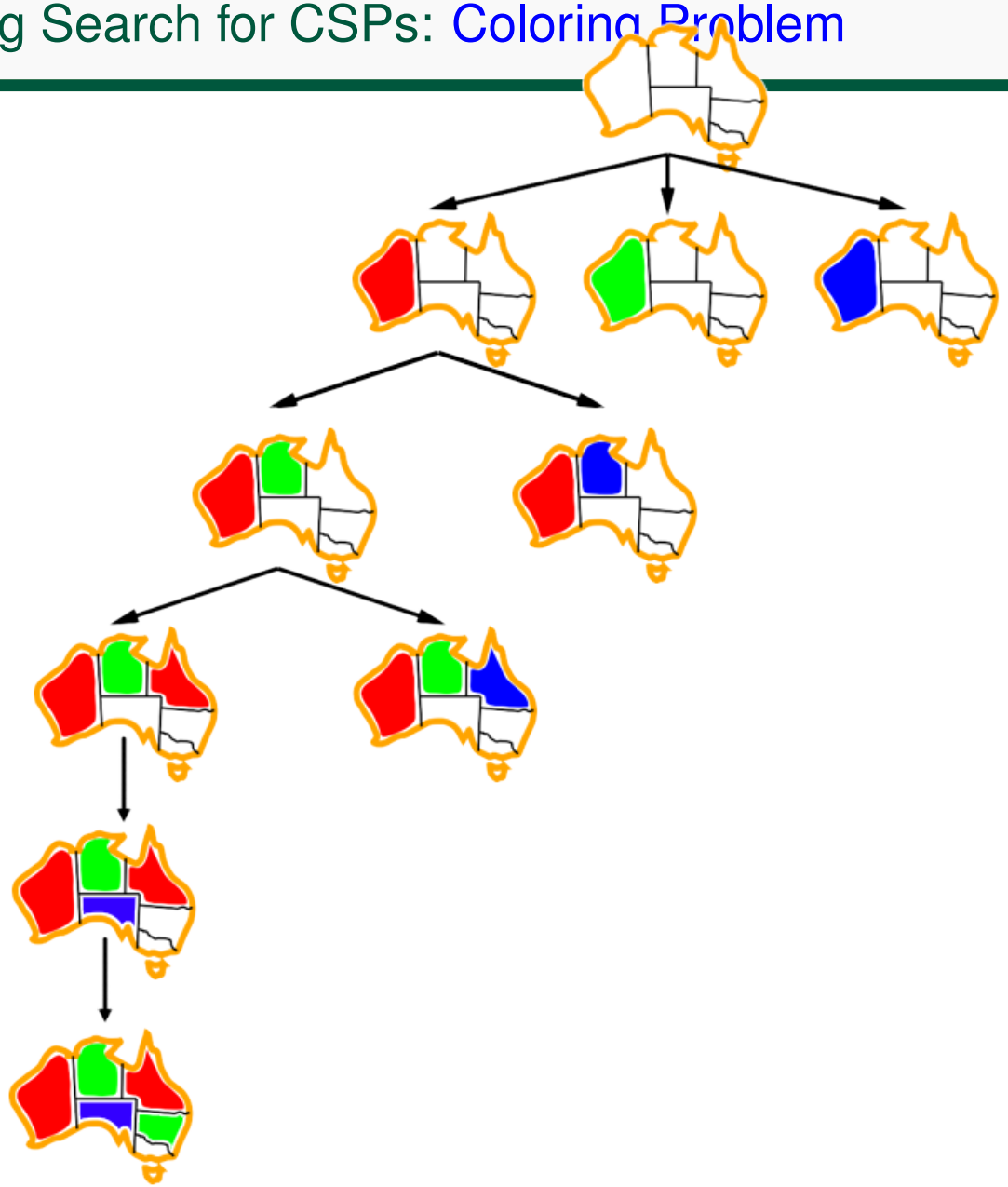


Figure: Trees(6) wrt. search(Backtracking) for `Coloring Problem`

Constraint Satisfaction Problems

Backtracking Search for CSPs: Colori blem

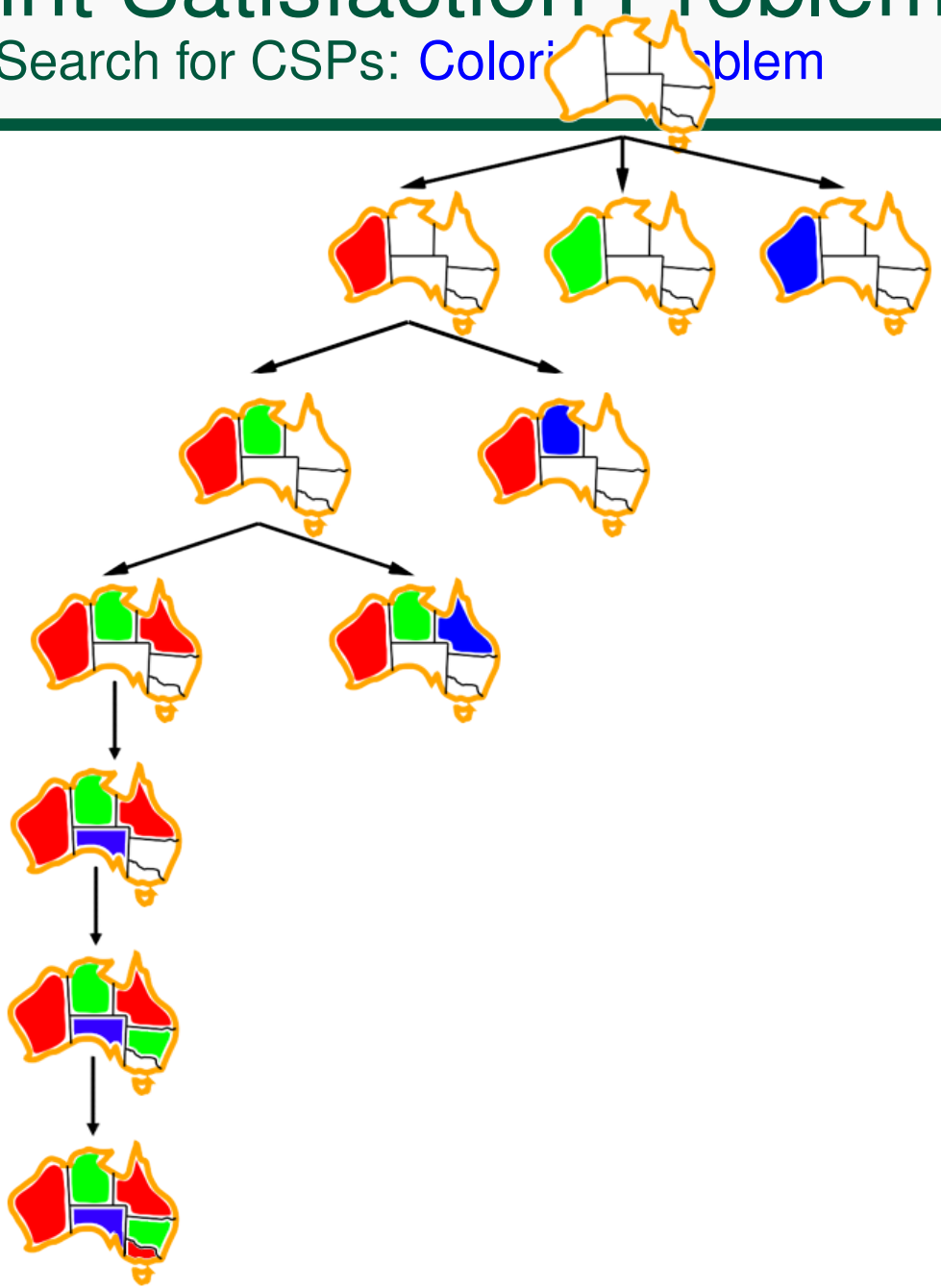


Figure: Trees(7) wrt. search(Backtracking) for Coloring Problem

Constraint Satisfaction Problems

Backtracking Search for CSPs: Coloring Problem

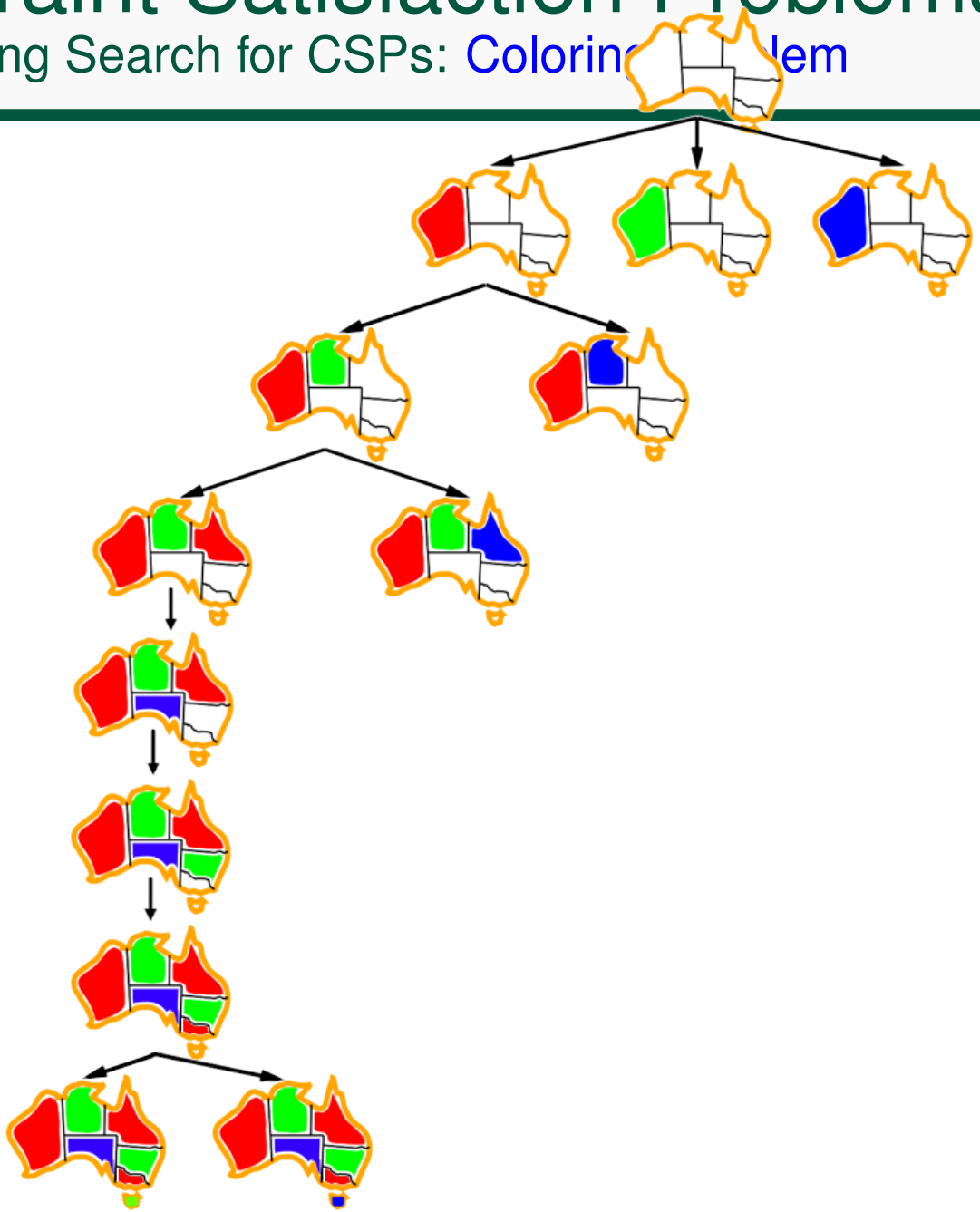
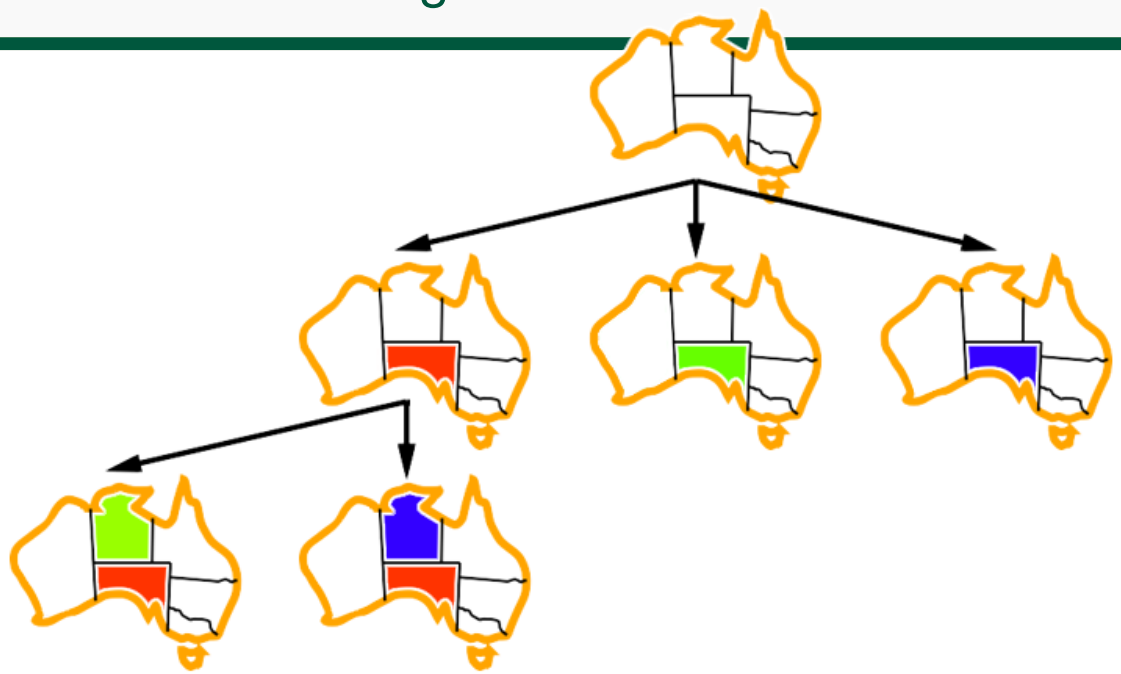


Figure: Trees(8) wrt. search(Backtracking) for Coloring Problem

Constraint Satisfaction Problems
Optimization of Backtracking Search for CSPs



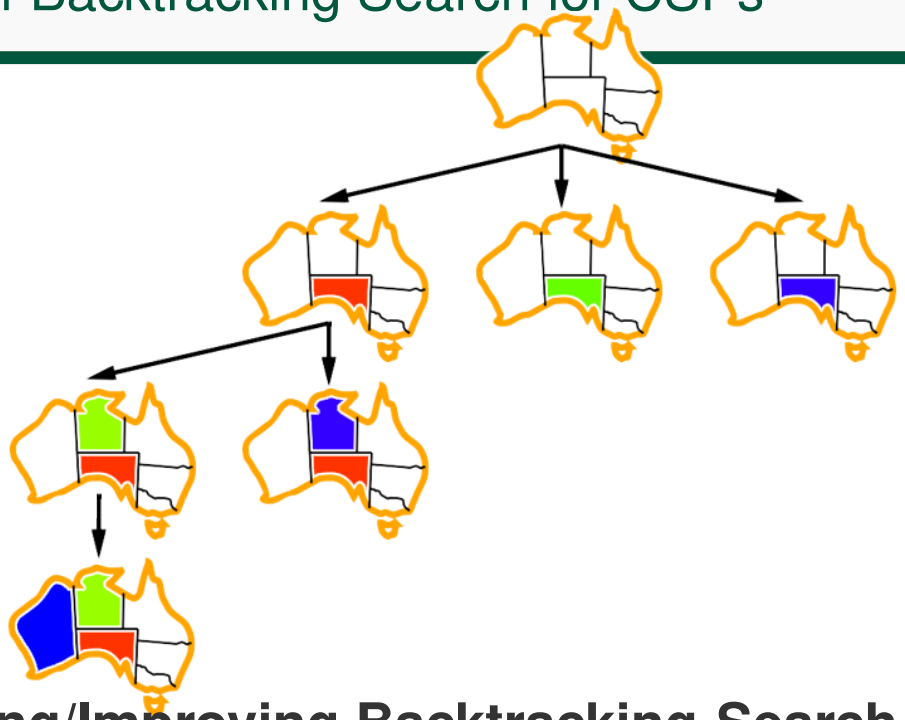
- **Optimizing/Improving Backtracking-Search algorithm:**

  How do we `infer/decide` the variable/node(Next) to process wrt. a CSP?

(1.1) Degree Heuristics: selects the variable, X_i, involved in the number(Largest) of constraints wrt. other unassigned variables, $X_j \ldots X_n$. Aids in choosing the variable/node(Start or First) wrt. a CSP.

- Example: We have chosen to start-off with South Australia, because it has the highest **Degree Heuristics**. Other regions/variables are interacting or bordering with 2 to 3 regions/variables, but South Australia is interacting or bordering with exactly 5 regions/variables. Next is Northern Territory based on **Degree Heuristics** too.

Constraint Satisfaction Problems
Optimization of Backtracking Search for CSPs



- **Optimizing/Improving Backtracking-Search algorithm:**

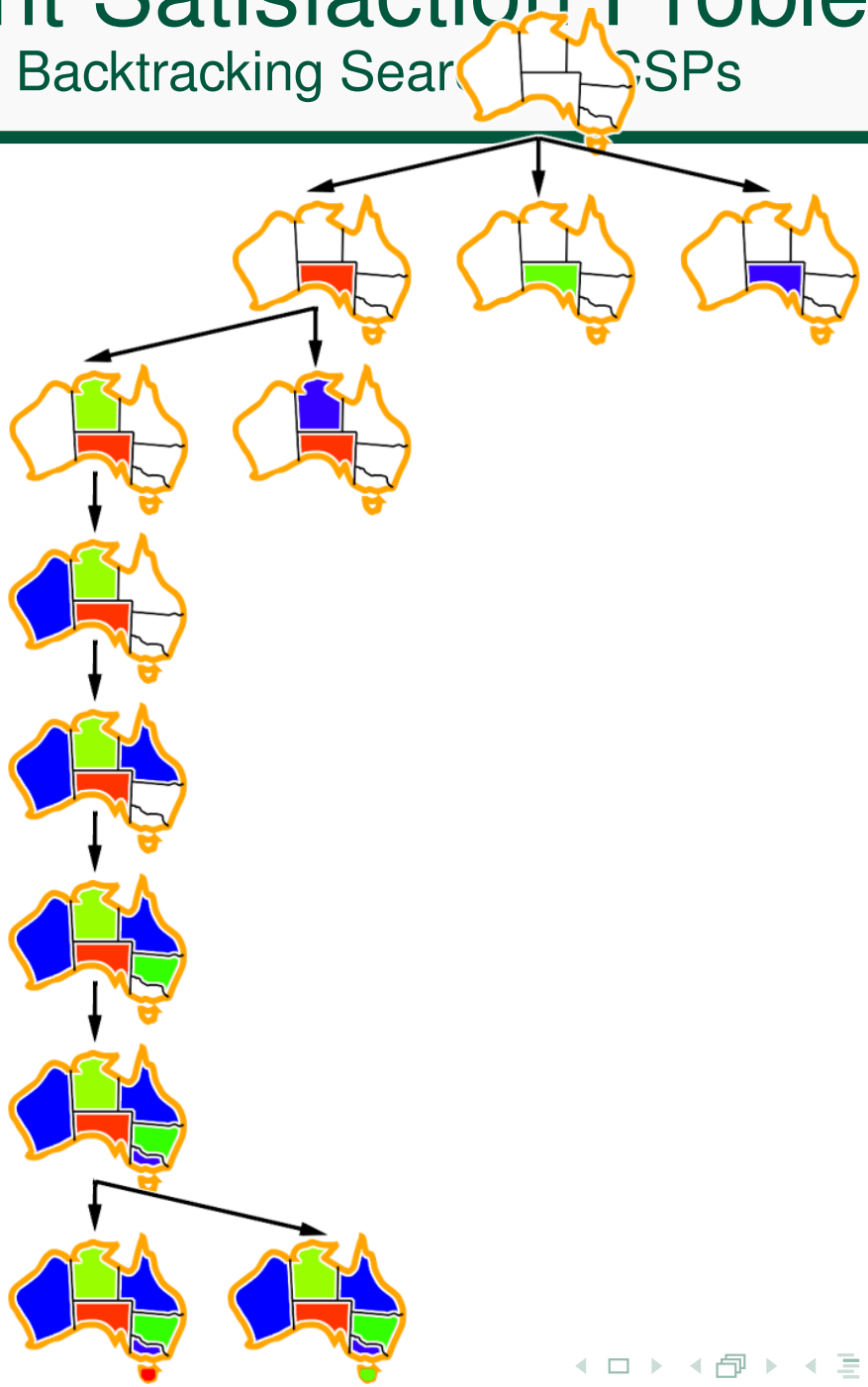
  How do we `infer/decide` the variable/node(Next) to process wrt. a CSP?

(1.2) Minimum Remaining Value (MRV): selects the variable, X_i, with the `fewest` permissible values, $v_m \in D_i$, remaining wrt. the predefined constraints. Also known as: Most Constrained Variable (MCV) or Fail-First Heuristic because it picks a variable, X_i, that is most likely to cause a *failure* early.

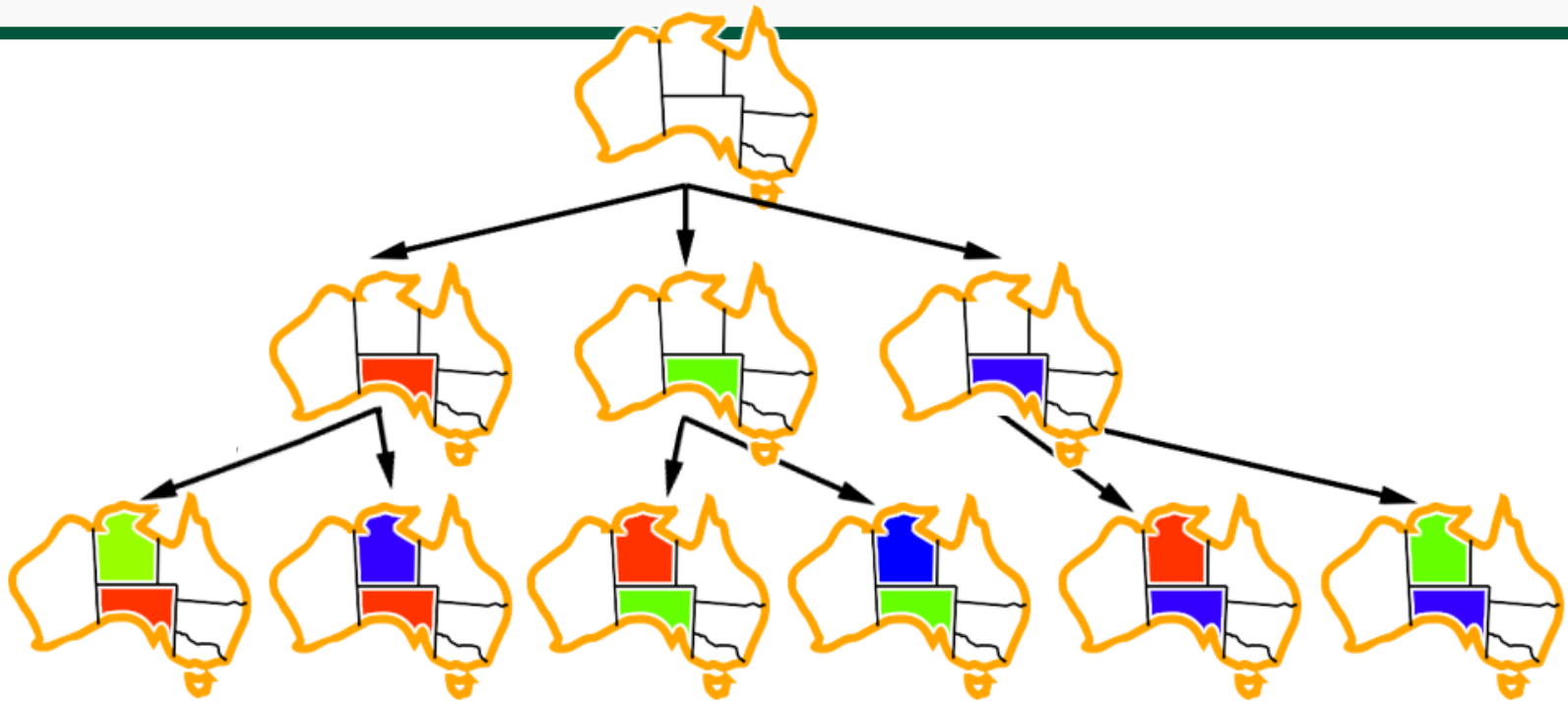
- Example: We chose next region/variable as: Western Australia based on **Minimum Remaining Value (MRV)**. Thus, Western Australia & Queensland are the next regions/variables with the lowest **MRV** wrt. permissible values, $v_m \in D_i$, for assignment.

Constraint Satisfaction Problems

Optimization of Backtracking Sear CSPs



Constraint Satisfaction Problems
Optimization of Backtracking Search for CSPs



- **Optimizing/Improving Backtracking-Search algorithm:**

  How do we `infer/decide` the variable/node(Next) to process wrt. a CSP?

(1.3) Least Constraining Value (LCV): it selects a value, $v_m \in D_i$, to assign to a variable, X_i, such that this value, v_m, leaves the *most possibilities open* for other unassigned variables, $X_j \ldots X_n$, interacting with assigned variable, X_i.

- Example: If we start by choosing a value(Red) for South Australia, it leaves at most two (2) values(Blue & Green) for the next region/variable(Northern Territory). Same applies if we start by choosing value(Green or Blue) for South Australia.

Constraint Satisfaction Problems
Optimization of Backtracking Search for CSPs

Algorithm 2 ForwardChecking() wrt. Backtracking-Search algorithm

```
 1: function FORWARDCHECKING(csp)
 2:     for each var-X_i in csp do
 3:         for each unassigned-var-X_j linked to var-X_i do
 4:             for each value-v_m in domain, D_j → X_j do
 5:                 if isConsistent(value-v_m, unassigned-var-X_j) == 'false' then
 6:                     (D_j → X_j) − v_m
 7:                 end if
 8:             end for
 9:         end for
10:     end for
11: end function
```

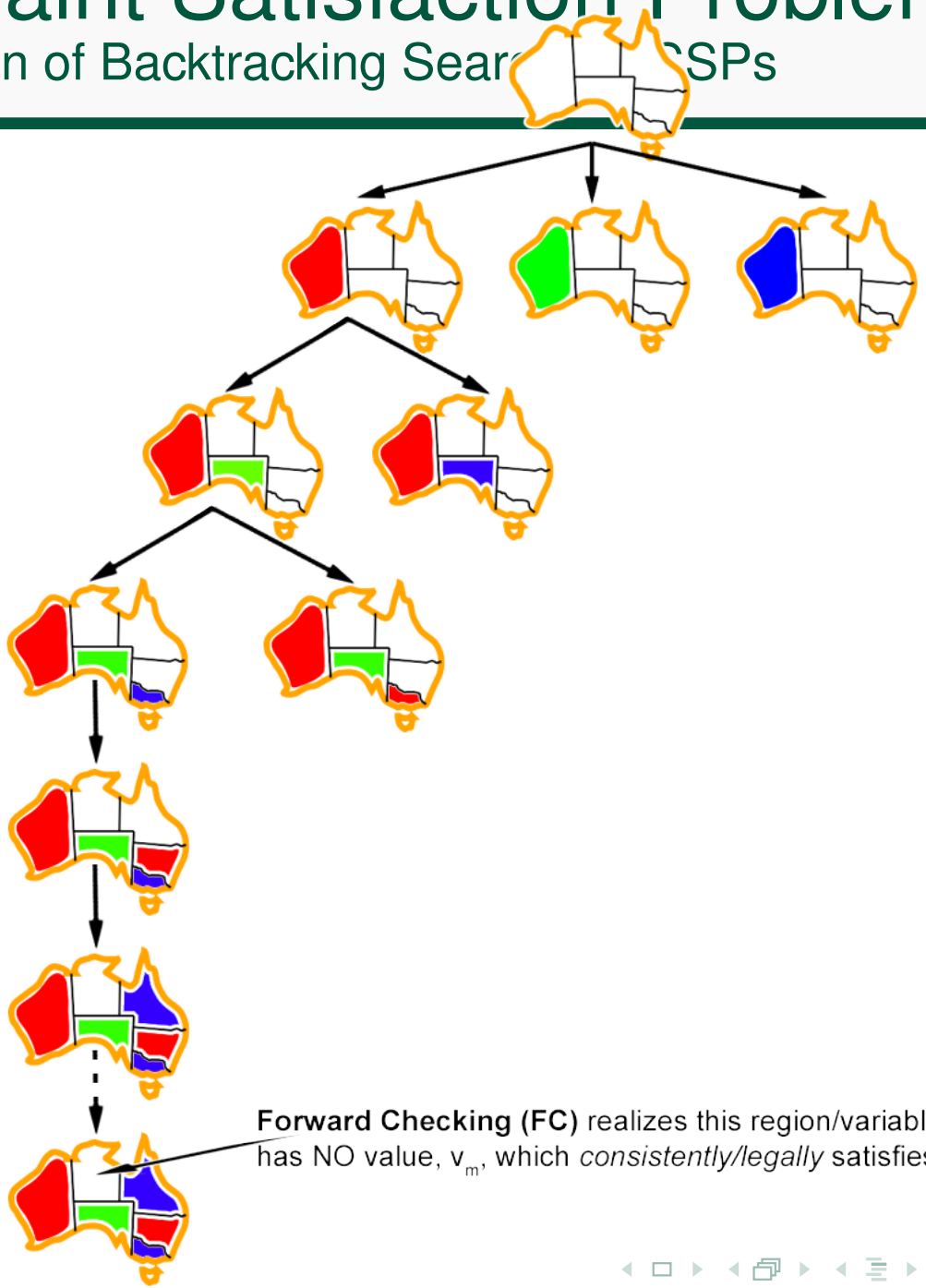
- **Optimizing/Improving Backtracking-Search algorithm:**

  How to `infer/decide` a value from a domain, $v_m \in D_i$, to assign(Consistent) to a variable/node, X_i, wrt. a *linked* unassigned-neighbor variable/node, X_j?

(2.0) Forward Checking (FC): from node(Root), via search(DepthFirst), and after assigning a value, $v_m \in D_i$, to a variable, X_i; we ALWAYS forward-check to ensure unassigned-neighbor variable/node, X_j, has a value, $v_m \in (D_j \rightarrow X_j)$, that will be assignment(Consistent/Legal) wrt. CSP.

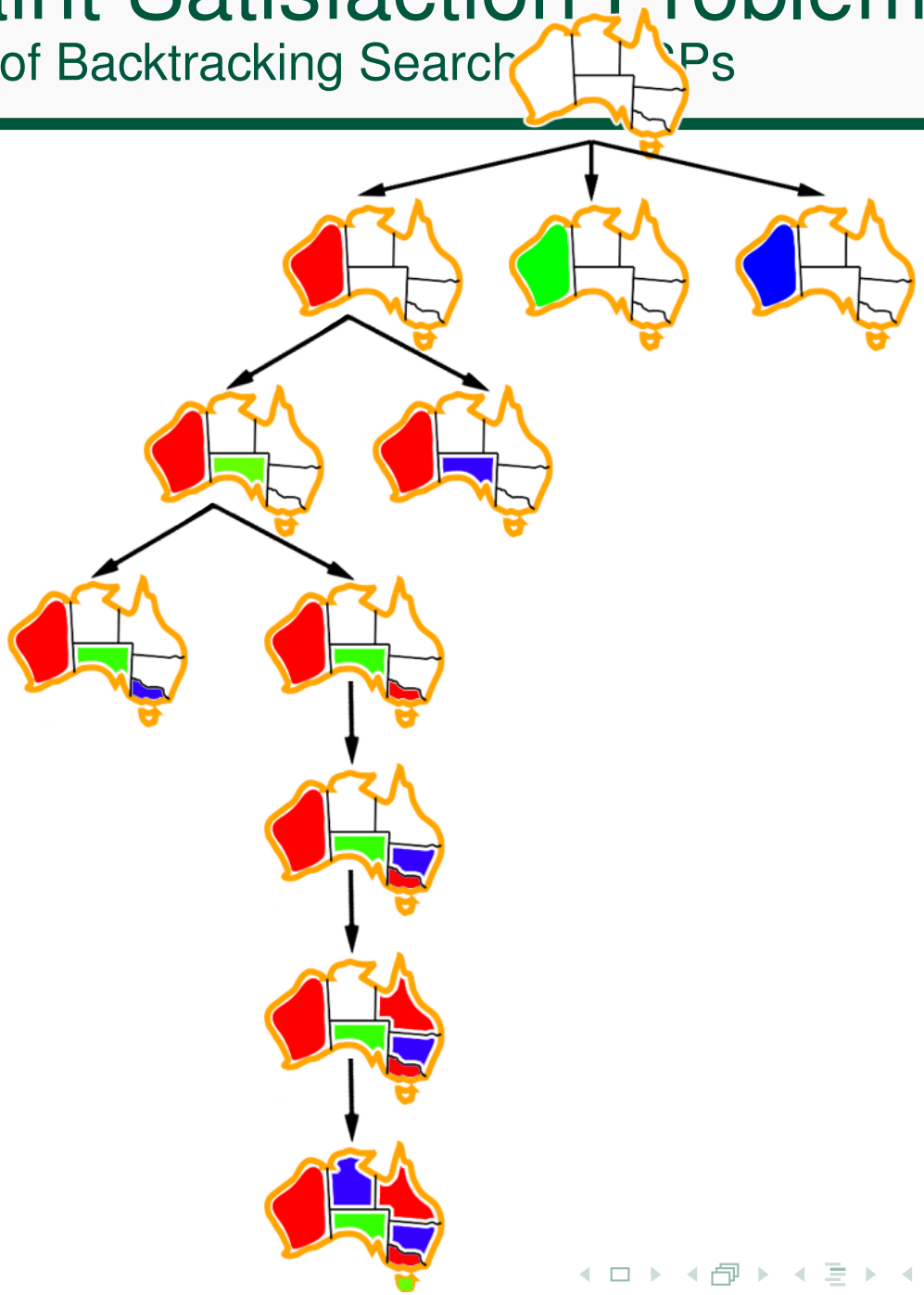
Constraint Satisfaction Problems

Optimization of Backtracking Sear SPs



Constraint Satisfaction Problems

Optimization of Backtracking Search CSPs



Local Search
Introduction

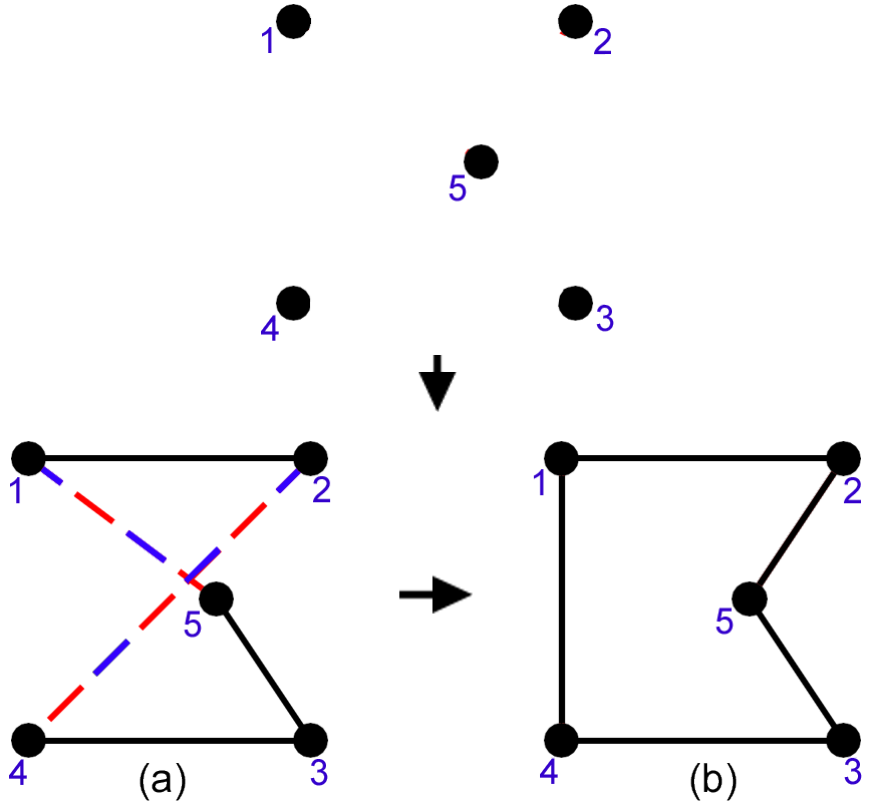
- In many problems(Search & Optimization), paths to the state(Goal) is NOT relevant.
  - In these problems(Search & Optimization), we only care about state(Goal) because it's the solution.

- **search(Local):**
  - It involves finding a final & valid configuration wrt. reaching state(Goal); and thus, the path/steps to this configuration is usually unimportant.
  - Local Search: search(Informed/Heuristics) + search(Goal-driven)
  - Local Search: scans from a state(Start/Initial) to the state(Neighboring), without keeping track of the paths, nor the set of states that were visited. It is NOT systematic; and in some cases, it may never explore portion(s) of the overall *search* space where a solution actually resides.

Local Search
Introduction

- **search(Local):**
  - Uses a space(State) = {set of `valid/complete` configurations}; and the aim is to select/choose the optimal `valid/complete` configuration from this space(State) set.
  - Major advantages:
    (1) uses very little memory;
    (2) finds reasonable solutions in a large or infinite space(State) where systematic algorithms are unsuitable.
  - Examples include, viz: Traveling Salesman Problem (TSP), job-shop scheduling, telecoms network optimization, etc.
  - search(Local) *employed* problems(Constraint Satisfaction)

Local Search
Examples: Traveling Salesman Problem (TSP)

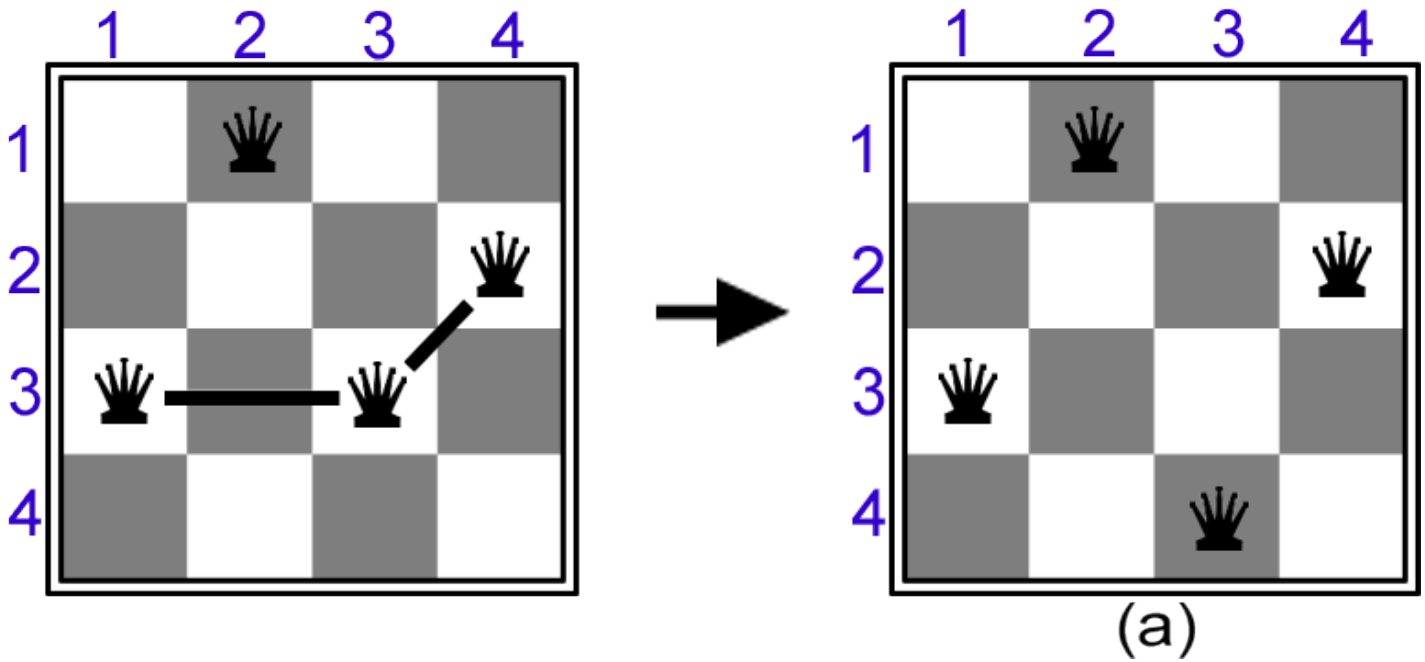


- **Domains where search(Local) can be employed:**

(1) Traveling Salesman Problem (TSP): traveling from a city/node(Start/Initial) and finding the `shortest` possible route that visits each city/node EXACTLY & ONLY once, and returns to the origin city/node(Start/Initial).

- Solution: {1, 2, 4, 3, 5, 1}, {1, 5, 3, 4, 2, 1}, {1, 2, 5, 3, 4, 1}

Local Search
Examples: *n*-Queens Problem



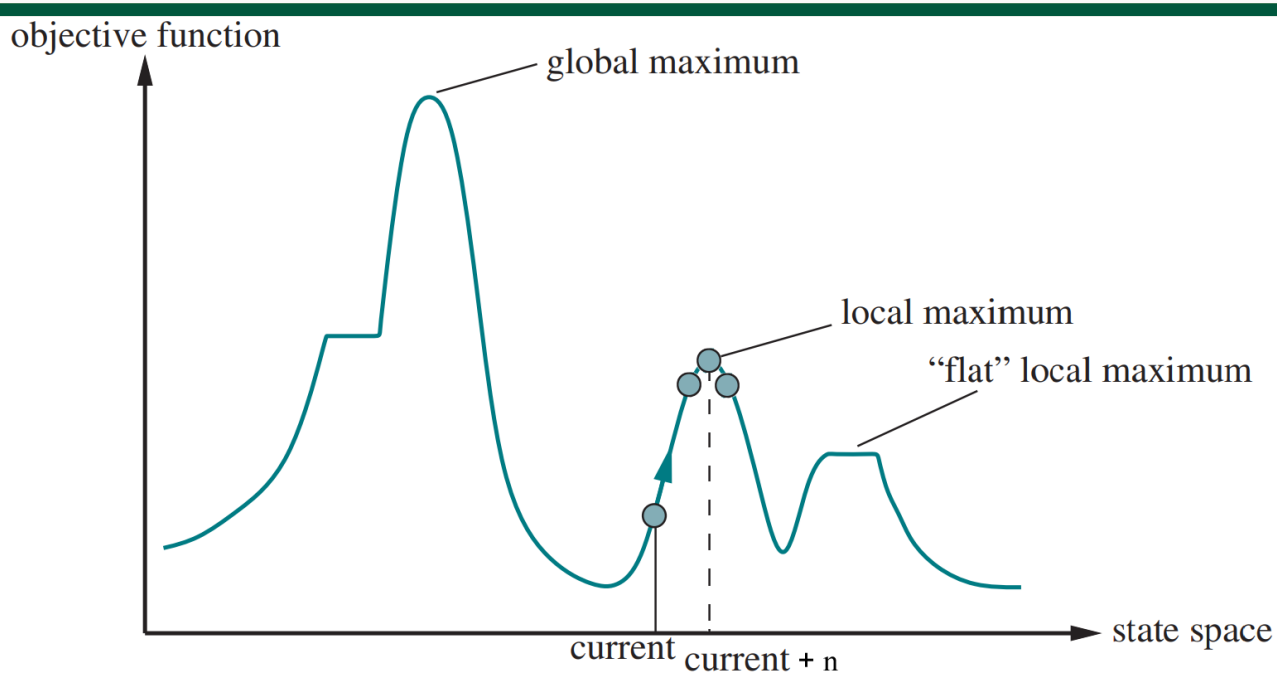
- **Domains where search(Local) can be employed:**

(2) *n*-Queens Problem: arrange *n*-count of chess(Queen) on $n \times n$ chessboard such that NO chess(Queen) can attack or defeat another.

- Finds a configuration or arrangement where NO **2** chess(Queen) are on the **same** `row`, `column`, or `diagonal`.
- Solution: move `(3,3)` one-step downwards to `(4,3)` as shown in (a).

Local Search

Variant 1: Hill-Climbing Search or Greedy-Local Search



- **Hill-Climbing/Greedy-Local Search:** search(Local)
  - It is type/variant of search(Local) which can be executed iteratively. For each iteration, the state(Start/Initial) is randomly chosen.
  - Keeps track of ONLY **1** state(Current) wrt. each iteration; and during each iteration, it moves forward to state(Neighbor: Left/Right) with `highest` value.
  - Each iteration threads the direction that provides the highest ascent.
  - Each iteration terminates when the state(Current) reaches its **Peak** - where NO state(Neighboring: Left or Right) has a *higher* value.

Local Search

Variant 1: Hill-Climbing Search or Greedy-Local Search

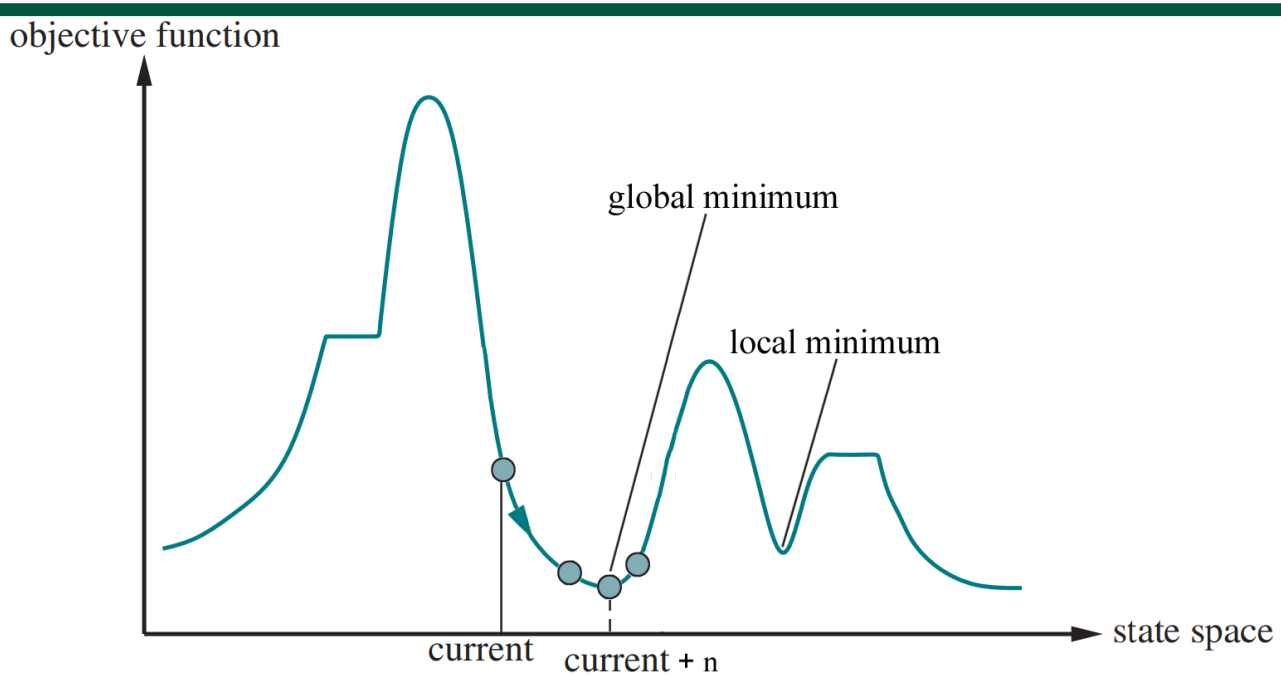
Algorithm 3 Hill-Climbing Search algorithm

```
1:  function HillClimbing(searchSpace)
2:      currentState = initialState(searchSpace)          ▷ Randomly chosen
3:      repeat:
4:          neighborState = bestNeighbor(LeftNeighbor, RightNeighbor)
5:          if value(neighborState) > value(currentState) then
6:              currentState = neighborState
7:          else                          ▷ value(neighborState) ≤ value(currentState)
8:              return currentState                          ▷ Local Maximum
9:          end if
10:
11:     until currentState == PEAK/HIGHEST
12: end function
```

- **Hill-Climbing or Greedy-Local Search:** search(Local)
  - It employs a function(Fitness/Utility) as its function(Objective) - where the goal is to maximize this function.
  - NOTE: It does NOT look ahead beyond the immediate state(Neighboring: Left or Right) wrt. state(Current).
  - NOTE: It does NOT backtrack for any reason, because it does NOT keep track of past states or paths.

Local Search

Variant 2: Gradient-Descent Search or Greedy-Local Search



- **Gradient-Descent/Greedy-Local Search:** search(Local)
  - It is type/variant of search(Local) which can be executed iteratively. For each iteration, the state(Start/Initial) is randomly chosen.
  - Keeps track of ONLY **1** state(Current) wrt. each iteration; and during each iteration, it moves forward to state(Neighbor: Left/Right) with `lowest` value.
  - Each iteration threads the direction that provides the lowest descent.
  - Each iteration terminates when the state(Current) reaches its **Trough** - where NO state(Neighboring: Left or Right) has a *lower* value.

Local Search

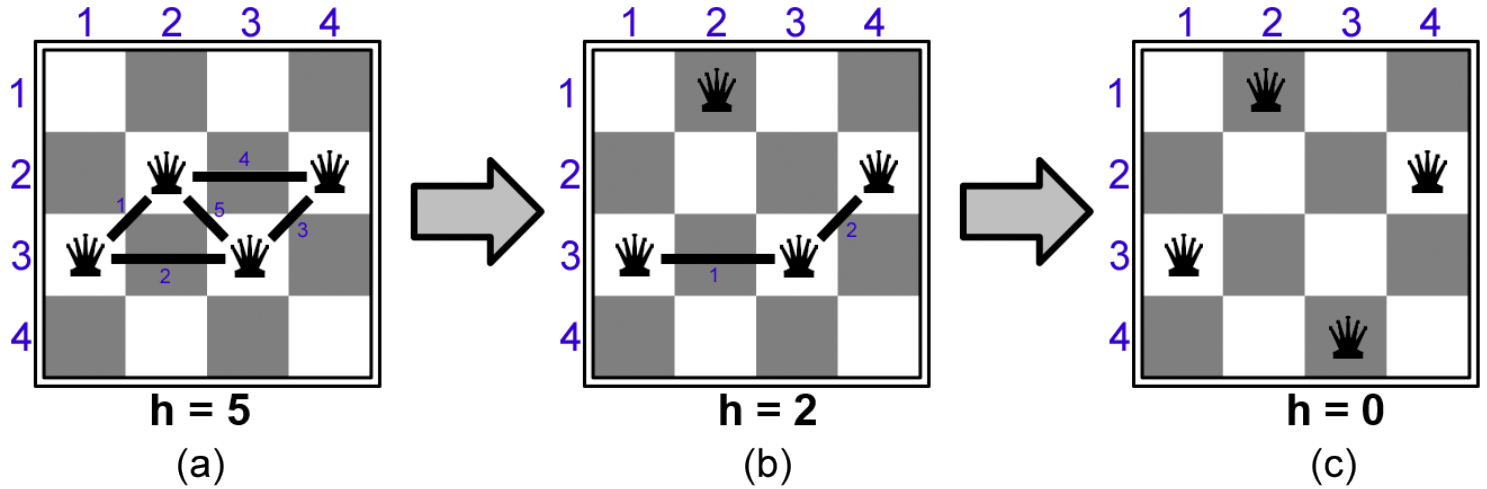
Variant 2: Gradient-Descent Search or Greedy-Local Search

Algorithm 4 Gradient-Descent Search algorithm

```
1:  function GradientDescent(searchSpace)
2:      currentState = initialState(searchSpace)          ▷ Randomly chosen
3:      repeat:
4:          neighborState = worstNeighbor(LeftNeighbor, RightNeighbor)
5:          if value(neighborState) < value(currentState) then
6:              currentState = neighborState
7:          else                          ▷ value(neighborState) ≥ value(currentState)
8:              return currentState                                ▷ Local Minimum
9:          end if
10:
11:     until currentState == TROUGH/LOWEST
12: end function
```

- **Gradient-Descent or Greedy-Local Search:** search(Local)
  - It employs a function(Cost/Loss) as its function(Objective) - where the goal is to minimize this function.
  - NOTE: It does NOT look ahead beyond the immediate state(Neighboring: Left or Right) wrt. state(Current).
  - NOTE: It does NOT backtrack for any reason, because it does NOT keep track of past states or paths.

Local Search
Examples: *n*-Queens Problem



- **Solving *n*-Queens Problem by Gradient-Descent/Greedy-Local Search:**

(3) 4-Queens Problem: arrange 4 chess(Queens) on 4×4 chessboard such that NO chess(Queen) can attack or defeat another.
NB: h = function(Heuristic) = pairs of chess(Queens) attacking each other.

- Solution: Find a configuration/arrangement where NO **2** chess(Queen) are on the **same** `row`, `column`, or `diagonal`.

(a) $\rightarrow$ (b): move `(2,2)` one-step upwards to `(1,2)` as shown in (b), and h minimizes to 2.

(b) $\rightarrow$ (c): move `(3,3)` one-step downwards to `(4,3)` as shown in (c), and h minimizes to 0. `Global/Local Minimum` is attained.

Local Search
Examples: *n*-Queens Problem

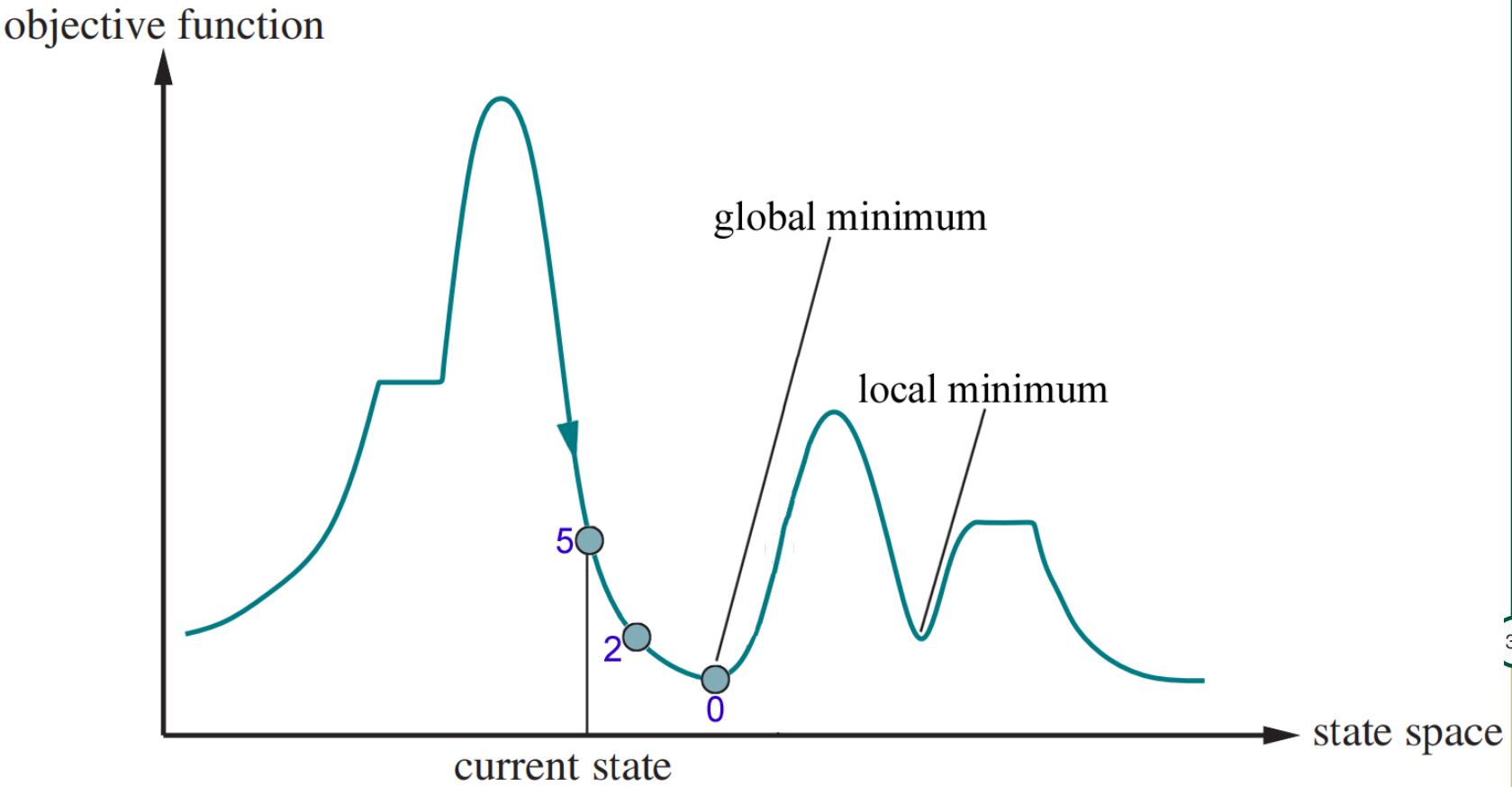
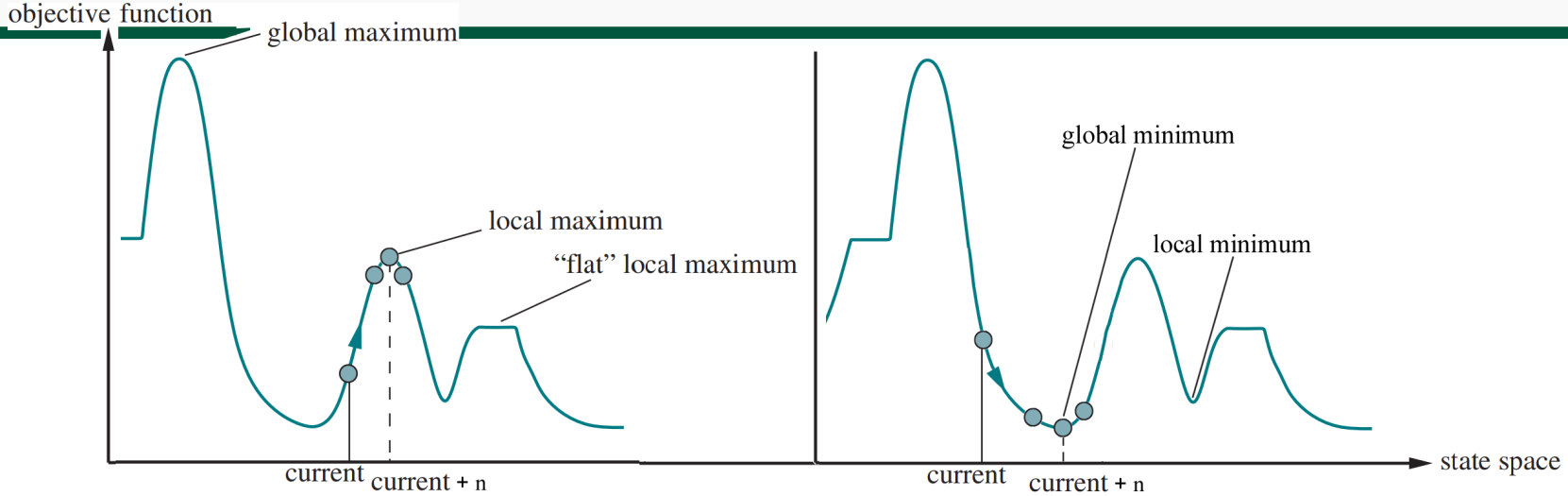


Figure: 4-Queens Problem above resolved via Gradient-Descent Search or Greedy-Local Search

Local Search

Issues/Bottlenecks wrt. Greedy-Local Search



- **Issues wrt. search(Hill-Climbing or Gradient-Descent)**:
  (1) Local Maxima in Hill-Climbing OR Local Minima in Gradient-Descent:
  - **Solution:** Random-Restart Hill-Climbing/Gradient-Descent.

  This overcomes `local maxima/minima` by conducting iterations of search (Hill-Climbing or Gradient-Descent) from random-chosen state(Start/Initial). Thereafter, the goal(Global Maxima or Global Minima) is selected as the maximum or minimum, respectively, wrt. all iterations.

  (2) Plateaus in Hill-Climbing OR Gradient-Descent: may be a `flat Local Maxima` wrt. Hill-Climbing OR a `flat Local Minima` wrt. Gradient-Descent.
  - **Solution:** Random/Consecutive Sideways Moves.

  Overcomes `plateaus` by moving sideways, consecutively, for a predefined no. of times until we escape this `plateau`.

Local Search
Variant 3: Simulated-Annealing Search

Algorithm 5 Simulated-Annealing Search algorithm

```
1:  function SIMULATEDANNEALING(searchSpace)
2:      currentState = initialState(searchSpace)          ▷ Randomly chosen
3:      for t = 100 to 0 do
4:          tmprt = t
5:          if tmprt == 0 then                            ▷ Local Maximum
6:              return currentState
7:          else
8:              neighborState = randomNeighb(LeftNeighbor, RightNeighbor)
9:              Energy = value(neighborState) − value(currentState)
10:             if Energy > 0 then                        ▷ Good Move: Forward Direction
11:                 currentState = neighborState
12:             else                                      ▷ BAD Move: Backward Direction, but on probability
13:                 prob = exp^(Energy/tmprt)
14:                 if prob > 0.5 then                    ▷ Threshold = 0.5 wrt. accept/reject 'BAD move'
15:                     currentState = neighborState
16:                 end if
17:             end if
18:         end if
19:     end for
20: end function
```

Local Search
Variant 3: Simulated-Annealing Search

- **Simulated-Annealing Search:** search(Local)
  - Inspired by concept(Metallurgy): process used to temper/harden metals & glass by heating them to a `high temperature`, and then gradually cooling them - this allows the metal or glass to reach a low-energy crystalline state.
  - It is type/variant of search(Local) which is executed iteratively wrt. a temperature, T, value. Usually begins with a large temperature, T, value. **NB:** Temperature, T, can be a function(Time).
  - During each iteration, the temperature, T, value decreases; and this decreases the value(Probability) too.
  - Keeps track of ONLY **1** state(Current) wrt. each iteration; and during each iteration, it moves to either of its state(Neighbor: Left/Right) based on a value(Probability).
  - The procedure terminates when the temperature, T, value reaches 0. Thus, at 0, it usually corresponds to a **Peak** state - where NO state (Neighboring: Left or Right) has a *higher* value.
- escape `local maxima` by allowing some bad moves to a state(Neighboring: Left or Right) with lower value, but gradually decrease their size and frequency of these bad moves.

Local Search
Variant 3: Simulated-Annealing Search

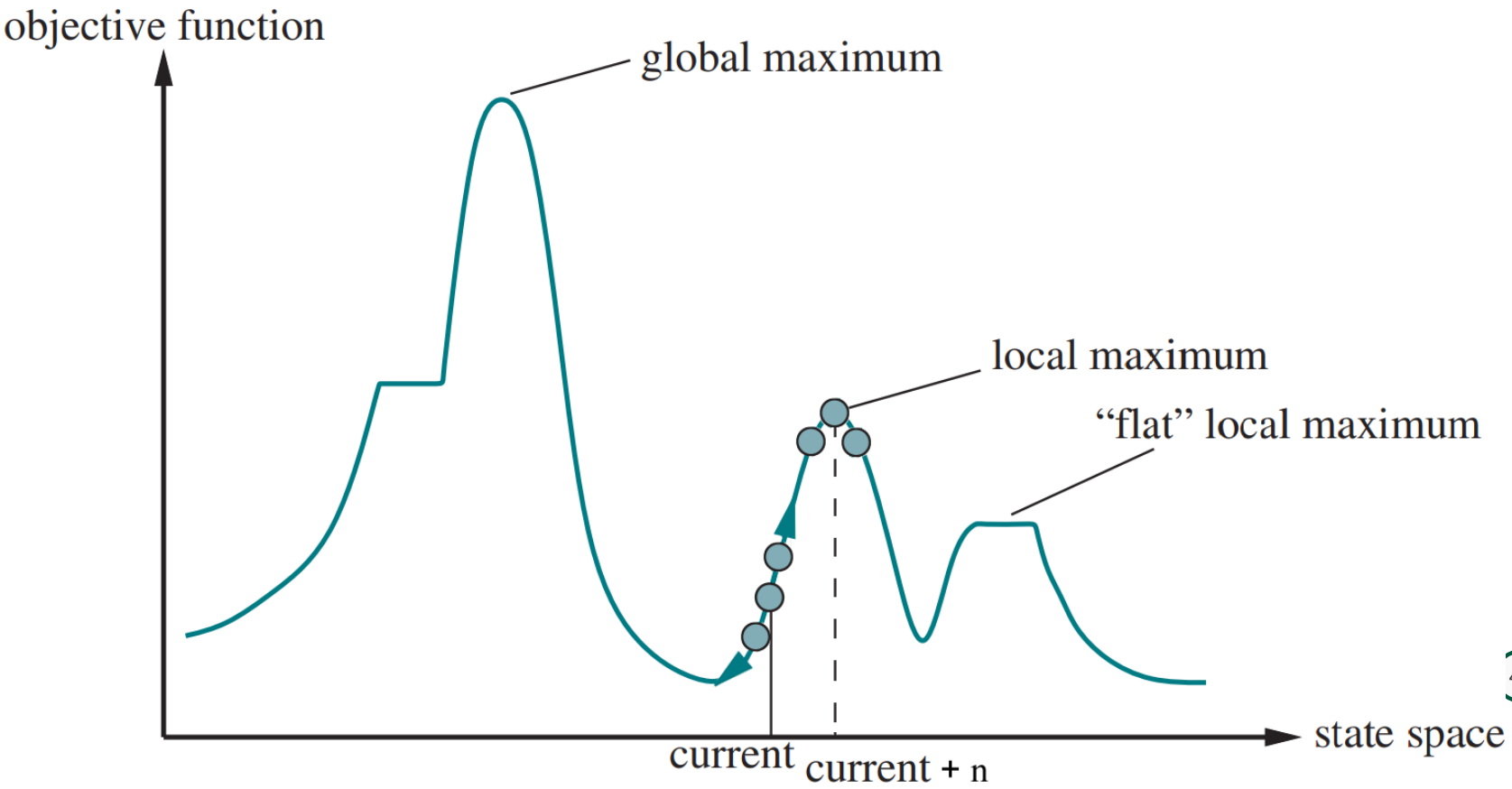


Figure: Simulated-Annealing Search

In-class Activity

CSPs & Local Search
Class/Game Activity

1. **An assignment of values, $v_m \in D_i$, from a domain to a given variable, X_i, wrt. a constraint in CSP is known as?**
   A. `Complete` assignment
   B. `Legal` assignment
   C. `Partial` assignment
   D. None of the above

2. **A set of "action-state-cost" triples which informs a algorithm(Search) on what should be its next move(s) is referred to as?**
   A. `Heuristic` function
   B. `Evaluation` function
   C. `Successor` function
   D. None of the above

CSPs & Local Search

Class/Game Activity

3. **A goal-driven strategy(Search) which depends on the light of the function(Evaluation/Heuristic) without tracking(Paths or States) is known as?**
   A. A* (Best-First) Search
   B. Iterative Deepening A* (Best-First) Search
   C. Greedy (Best-First) Search
   D. Local Search

4. **A goal-driven strategy(Search) which depends on the light of the function(Evaluation/Heuristic) AND with tracking(Paths or States) is known as?**
   A. A* (Best-First) Search
   B. Iterative Deepening A* (Best-First) Search
   C. Greedy (Best-First) Search
   D. All of the above

Constraint Satisfaction Problems
Class/Game Activity

5. **Explain the concept of search(Local)?**

6. **Define in details the concept of search(Hill-Climbing)?**

7. **Explain the concept of search(Gradient-Descent)?**

Questions?
&
Answers!