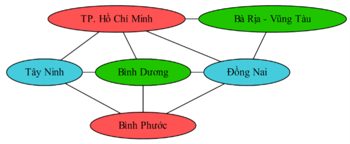
04-05 - Constraint Satisfaction Problems (CSPs)

Ví dụ bài toán ràng buộc: tô màu bản đồ



- Đồ thị ràng buộc: đồ thị 2 ngôi
- Ràng buộc 2 ngôi

- Số lượng biến: 6 tỉnh thành phố ĐNB
- Miền giá trị = {red, green, blue}
- Ràng buộc: các tỉnh thành phố (các đỉnh) cạnh nhau có màu khác nhau

Các thuật toán tìm kiếm cũ khi giải quyết CSP

- BFS: lời giải của CSP nằm ở tầng đáy ⇒ BFS thất bại rất nặng
- DFS: có thể có lời giải nhưng mất rất nhiều thời gian và không tối ưu

⇒ Cải tiến DFS → Backtracking

Backtracking

- Chiến lược:
    - idea 1: mỗi thời điểm chỉ xét 1 biến
    - idea 2: kiểm tra ràng buộc tại thời điểm xét

```
CSP-BACKTRACKING(PartialAssignment a)
    If a is complete then return a
    X <- select an unassigned variable
    D <- select an ordering for the domain of X
    For each value v in D do
        If v is consistent with a then
            Add (X = v) to a
            result <- CSP-BACKTRACKING(a)
            If result <> failure then return result
            Remove (X = v) from a
    Return failure
```

Nhận xét

- Chắc chắn tìm ra lời giải nếu lời giải có tồn tại
- Nếu bài toán có chứa lời giải tầm thường như knapsacks (lời giải thỏa mãn tất cả ràng buộc nhưng không có giá trị), backtracking có thể rơi vào kết thúc sớm trước khi tìm ra lời giải tối ưu.
- Vét cạn thay vì kết thúc sớm → tốn chi phí tính toán

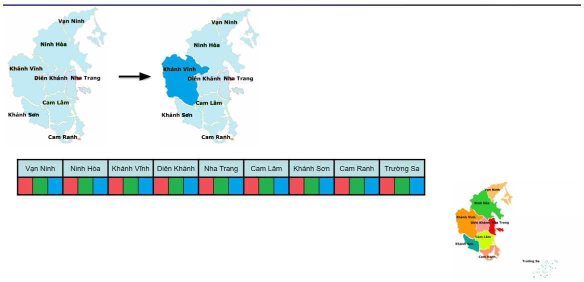
⇒ Để tìm ra lời giải tối ưu, phải vét cạn bằng DFS nhưng phải cải tiến khả năng xét vi phạm ràng buộc

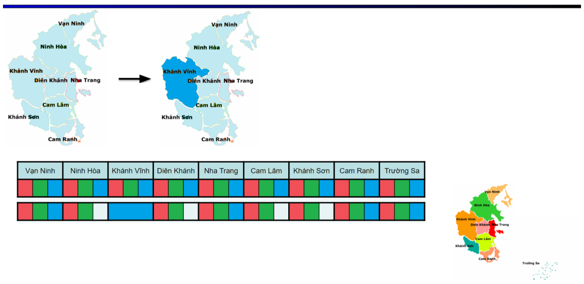
⇒ Filtering → Ordering → Structure

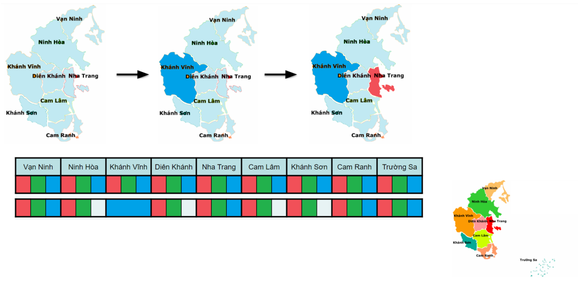
Filtering

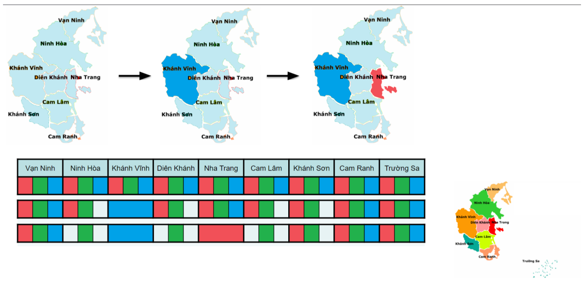
Forward Checking

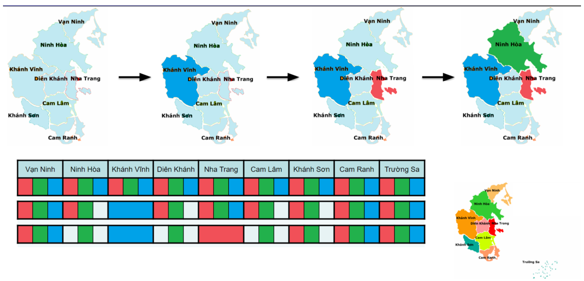
- Mỗi lần gán giá trị 1 biến → tra miền giá trị các biến khác → gạch bỏ các giá trị trong miền của biến khác nếu giá trị đó gây vi phạm ràng buộc

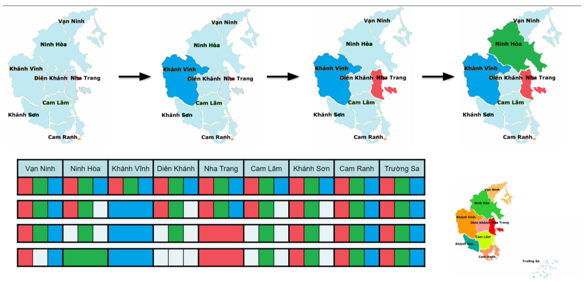












Diên Khánh mất màu → nhánh này không chứa lời giải

⇒ Forward checking giúp phát hiện nhánh không chứa lời giải vì nó giúp phát hiện hiện tượng mất miền dữ liệu

- Nhược điểm: khả năng cắt nhánh vẫn còn thụ động và phát hiện vi phạm trễ, vì lời giải trên thất bại không phải vì chọn Ninh Hòa là green mà do chọn Nha Trang red (do **toàn bộ** nhánh con của Ninh Hòa green đều thất bại nhưng thuật toán vẫn đi kiểm tra các nhánh con đó cho đến khi Diên Khánh mất màu)

→ Làm cách nào để phát hiện Nha Trang **red** là sai?

Constraint Propagation

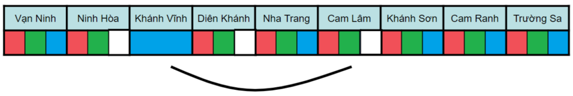
- Kiểm tra tính nhất quán của 1 cung
  - 1 cung $X \rightarrow Y$ là nhất quán khi với mọi giá trị trong miền giá trị X luôn luôn có 1 giá trị nào đó ở biến Y mà giúp cho phép gán tồn tại (thỏa mãn ràng buộc)

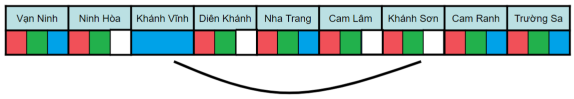
- Kiểm tra tính nhất quán 1 cung: Diên Khánh → các đỉnh khác

Ban đầu Khánh Vĩnh blue → kiểm tra tính nhất quán từng cung trỏ tới KV, nếu chưa nhất quán thì làm cho cung đó nhất quán

function $\texttt{Revise}(i, j)$
    change $:= \text{false}$
    for each $a \in d_i$ **do**
        if $\forall_{b \in d_j} \quad \neg c_{ij}(a, b)$ **then**
            change $:= \text{true}$
            remove a **from** d_i
    return change

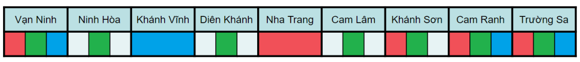
xóa biến trong miền giá trị i



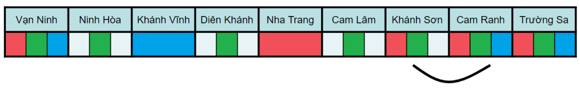


⇒ **chậm, phát hiện trễ** → phải kiểm tra tính nhất quán của toàn bộ đồ thị

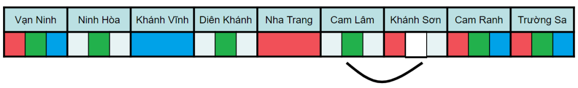
- Kiểm tra tính nhất quán toàn bộ đồ thị:



ban đầu

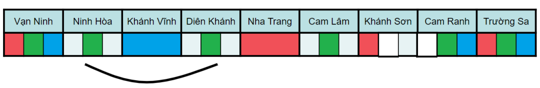


nhất quán 2 chiều



không nhất quán → xóa green KS

không nhất quán→xóa red CR



1 trong 2 NH DK bay màu

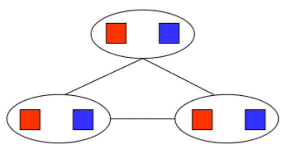
procedure $\texttt{AC-3}(X, D, C)$
    $Q := \{(i, j), (j, i) \mid \ c_{ij} \in C\}$ // each pair is added twice
    while $Q \neq \emptyset$ **do**
        $(i, j) := \texttt{Fetch}(Q)$ // selects and removes from Q
        if $\texttt{Revise}(i, j)$ **then**
            $Q := Q \cup \{(k, i) \mid \ c_{ki} \in C, k \neq j\}$

thêm cung (k,i) lại vào Q để kiểm tra ngược lại các trạng thái trước sau khi bỏ biến trong miền i

⇒ chi phí lớn vì phải kiểm tra lặp đi lặp lại

▼ Hạn chế AC3

- Nhất quán k ngôi: với mỗi k nodes, bất kì phép gán nào cho k-1 node thì node thứ k còn lại phải tồn tại phép gán hợp lệ



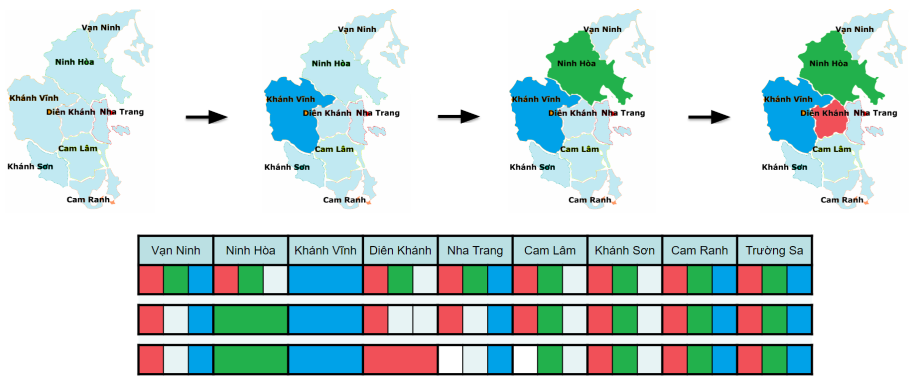
→ số ngôi càng tăng, phép gán càng phức tạp, thời gian cắt tỉa rất lâu

Ordering

- Sắp xếp cây tìm kiếm:
    - Chọn biến nào gán giá trị trước
    - Chọn hướng đi nào

Minimum Remaining Values

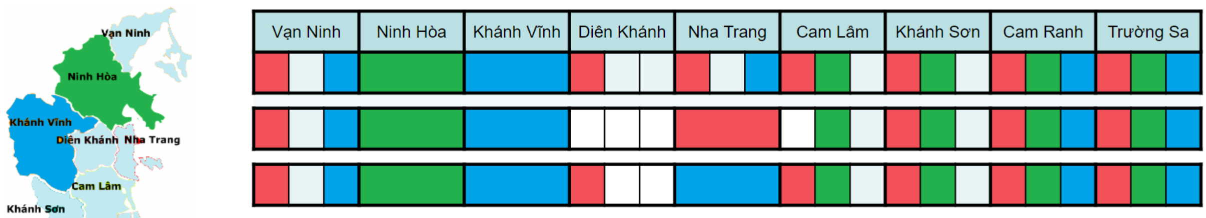
- Chọn biến có miền giá trị nhỏ nhất gán trước, là biến có số nhánh ít nhất → quay lui nhanh hơn



Ninh hòa green → Diên Khánh red ⇒ chọn miền giá trị nhỏ trước

Least Constraining Value

- Chọn biến → Chọn giá trị có ràng buộc ít nhất



Nha Trang red ảnh hưởng nhiều tới miền giá trị biến khác → khả năng cao nhánh con sẽ bị loại
Nha Trang blue ảnh hưởng ít hơn → chọn

Structure

- Khai thác cấu trúc đồ thị
- Khai thái bài toán con độc lập bằng cách tìm thành phần liên thông
- Ví dụ: giải bài toán có n=80, d=2, c=20
    - $2^{80} =$ 4 tỉ năm với 10 triệu nodes/giây
    - $(4)(2^{20}) = 0.4$ giây với 10 triệu nodes/giây

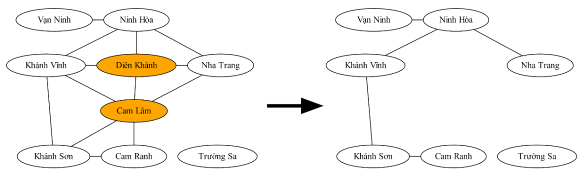
Đồ thị dạng cây

- Cầm bất kì node nào làm node gốc, các node còn lại sẽ tạo thành cây
- Nếu đồ thị ràng buộc không có chu trình (đồ thị dạng cây), CSP có thể được giải trong $O(nd^2)$, trong khi đồ thị tổng quát có thể lên đến $O(d^n)$

⇒ Hiếm

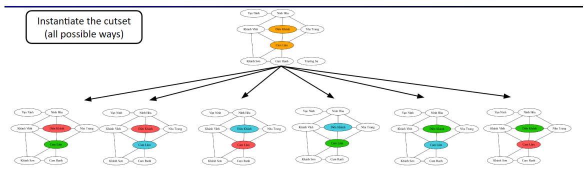
Đồ thị dạng gần cây - Nearly Tree-Structured CSPs

- Khi bỏ ít node (cắt node) trên đồ thị sao cho phần còn lại tạo cây

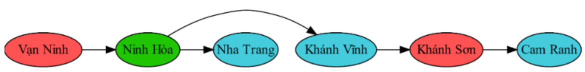


Cutset Conditioning

- Kĩ thuật điều kiện cắt node
- Steps:
  - Chọn tập cắt
  - Chọn toàn bộ cách tô màu hợp lệ cho tập cắt
  - Xóa miền giá trị của các biến liên quan trong cây
  - Giải cấu trúc cây bằng tính nhất quán của các cung (rất nhanh)



Xét TH DK red CL green → xóa miền giá trị biến liên quan



đồ thị cây sau khi cut đi 2 node Diên Khánh Cam Lâm, dấu → trỏ từ parent to child ⇒ giải rất nhanh

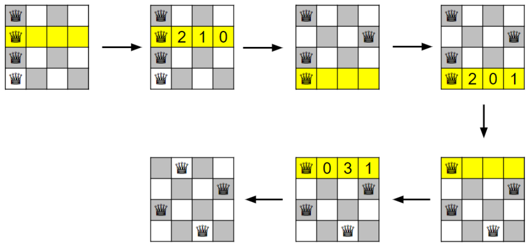
- Kiểm tra tính nhất quán từ lá đến đỉnh: tránh việc kiểm tra lại cung trước, quét 1 lượt là xong

Iterative Improvement/Local Search

- Gán giá trị ngẫu nhiên cho toàn bộ biến → kiểm tra biến nào vi phạm ràng buộc thì sửa từ biến đó

Thuật toán Min-Conflicts

- Kiểm tra miền giá trị của biến vi phạm ràng buộc, giá trị nào của miền khiến ràng buộc vi phạm ít hơn thì gán ngược lại giá trị đó cho biến



xét quân 2 đẩy sang cột 4 vì **0** vi phạm ràng buộc

- Di chuyển giữa những node lá lân cận cho tới khi nào nhảy tới node lá thỏa mãn ràng buộc → local search
- Nhận xét
    - Có thể không tìm ra lời giải vì chọn random
    - Nhiều khi xử lí vi phạm chỗ này lại gây ra vi phạm chỗ khác
    - Có thể giải n-queens 10 000 000
    - Cho dù chạy đến $\inf$ cũng có thể không tìm ra lời giải

Local Search

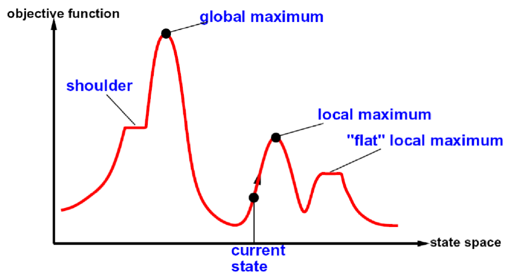
Thuật toán Hill Climbing

- Tìm tọa độ ứng với điểm cao nhất của dãy núi với điểm bắt đầu ngẫu nhiên → nếu hàng xóm cao hơn thì đi tới chỗ hàng xóm

⇒ dễ nhầm với cực đại địa phương

- Có thể random start tránh local maximum

```
function HILL-CLIMBING( problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node
   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
   end
```



- Có thể tăng bán kính lân cận, tuy nhiên khi tăng bán kính sẽ ảnh hưởng đến số trường hợp kiểm tra tăng lên → tốn chi phí tìm kiếm

Thuật toán Simulated Annealing

- Ban đầu nhiệt độ rất lớn, khi nhiệt độ giảm về 0 sẽ dừng lại → thuật toán leo đồi nhưng cho phép xuống đồi với xác suất ngẫu nhiên, xác suất qui định bởi nhiệt độ và chất lượng lời giải

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward steps

   current ← MAKE-NODE(INITIAL-STATE[problem])
   for t ← 1 to ∞ do
       T ← schedule[t]
       if T = 0 then return current
       next ← a randomly selected successor of current
       ΔE ← VALUE[next] – VALUE[current]
       if ΔE > 0 then current ← next
       else current ← next only with probability e^(Δ E/T)
```

- kiểm tra nhiệt độ
  - T=0 → trả về vị trí hiện tại *current*
  - T > 0 → kiểm tra lân cận *current* → chọn random 1 vị trí local → tính hiệu $\Delta E = Value(new) - Value(current)$
    - $\Delta E > 0$: *current = new* (vị trí mới tốt hơn cũ → dịch chuyển tới *new*)
    - $else$: *current=new* (vị trí mới tệ hơn vị trí hiện tại → dịch chuyển sang vị trí mới với xác xuất $e^{\frac{\Delta E}{T}} = \frac{1}{-e^{\frac{\Delta E}{T}}}$)
- Nhận xét:
  - Cây càng lớn → xác suất xuống đồi càng lớn
  - ΔE càng thấp → xs xuống đồi càng lớn
  - Trong lân cận, vị trí ngẫu nhiên mà chọn tệ → xs xuống đồi thấp, ngược lại thì vẫn chấp nhận xuống đồi
  - Khi bắt đầu chạy thì thuật toán cho phép xuống đồi với xs lớn → tính ngẫu nhiên cao ⇒ Nhờ xs xuống đồi, có khả năng thoát khỏi local max → tăng xs đến được global max
  - Nhiệt độ **T** giảm đủ chậm → chắc chắn tìm ra lời giải tối ưu
- Khuyết điểm:
  - Có thể chạy rất rất lâu vì thường sẽ mất nhiều bước xuống đồi nhưng xác suất mỗi lần xuống đồi thấp → để thoát được đồi thì xs cực kì thấp
- Trong thực tế khi huấn luyện neural network, không cần thiết tìm kiếm global max

  ⇒ Thường dùng để giải các bài toán khó không dùng NN