ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

IT3160E
Introduction to Artificial Intelligence

Chapter 3 – Problem solving

Part 5: Constraint Satisfaction problems

Lecturer:

Muriel VISANI

Acknowledgements:

Le Thanh Huong

Tran Duc Khanh

Department of Information Systems

School of Information and Communication Technology - HUST

Content of the course

- Chapter 1: Introduction
- Chapter 2: Intelligent agents
- Chapter 3: Problem Solving
    - Search algorithms, adversarial search
    - Constraint Satisfaction Problems
- Chapter 4: Knowledge and Inference
    - Knowledge representation
    - Propositional and first-order logic
- Chapter 5: Uncertain knowledge and reasoning
- Chapter 6: Advanced topics
    - Machine learning
    - Computer Vision

Outline

- Chapter 3 – part 1: un-informed (basic) algorithms
- Chapter3 - part 2: informed search strategies in graphs
- Chapter 3 – part 3: advanced search strategies
- Chapter 3 – part 4: adversarial search
- Chapter 3 – part 5: Constraint Satisfaction Problems
  - CSP introductive example
  - Definitions
  - Backtracking search
    - Choosing the next variable to assign: MRV and degree heuristic
    - Ordering the values to examine: LCV
    - Can we detect inevitable failure early? -> forward checking + a few words about arc consistency
  - Summary
  - Homework

Goal of this Lecture

Goal	Description of the goal or output requirement	Output division/ Level (I/T/U)
M1	Understand basic concepts and techniques of AI	1.2

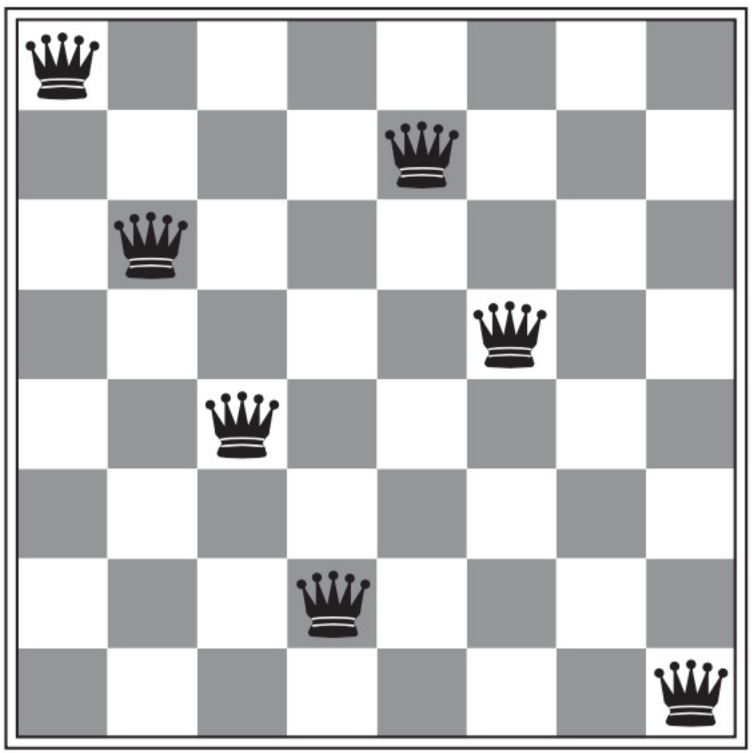
Constraint Satisfaction Problems

CSP introductive examples

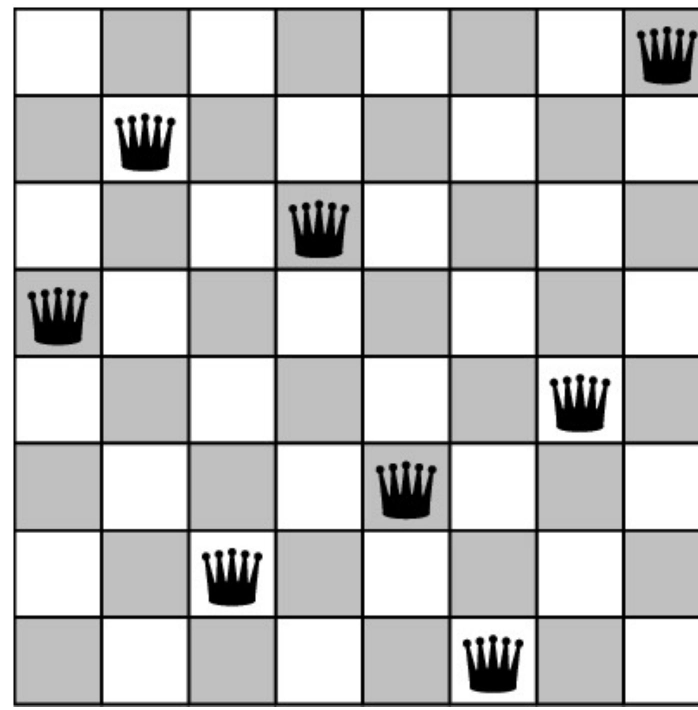
Introductive example #1

- **8-queens problem:**
  - place 8 queens on a chessboard such that no queen can attack any other

Configuration that does **not** meet the goal



Configuration that does **does** meet the goal



Introductive example #1

- **8-queens problem formulation using search algorithms:**
  - States: Any arrangement of 0 to 8 queens on the board
  - Initial state: No queens on the board
  - Actions: Add a queen to any empty square
  - Transition model: Returns the board with a queen added to the specified square
  - Goal test: 8 queens are on the board, none of them is attackable

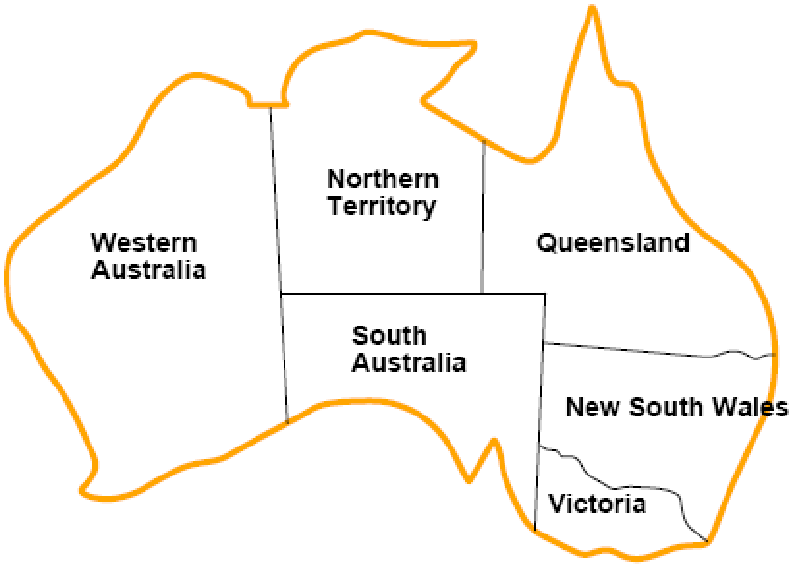
- **Difficulty:** Search graph is HUGE!!!
  - $64 \times 63 \times \ldots \times 57 = 64! / 56! \approx 1.8 \times 10^{14}$ possible sequences to investigate!!!

Introductive example #1

- Why is the search graph so huge when using search algorithms?
  - Because the states are defined as « Any arrangement of 0 to 8 queens on the board »
    - Each state is atomic (not "divisible")
- Idea of CSP algorithms
  - Each state is divided into *n* variables X_i with value in domain D_i
  - The goal test is a set of constraints over these variables
  - In the case of the 8-queen problem, the variables $Q_1,...,Q_8$ are the positions of each queen in columns 1,...,8 and each variable has the domain $D_i = \{1,2,3,4,5,6,7,8\}$.
    - Positions of queens in each column are enough because 2 queens cannot be in a same column
- Using that idea, CSP algorithms can solve a wide variety of problems **more efficiently** than search algorithms

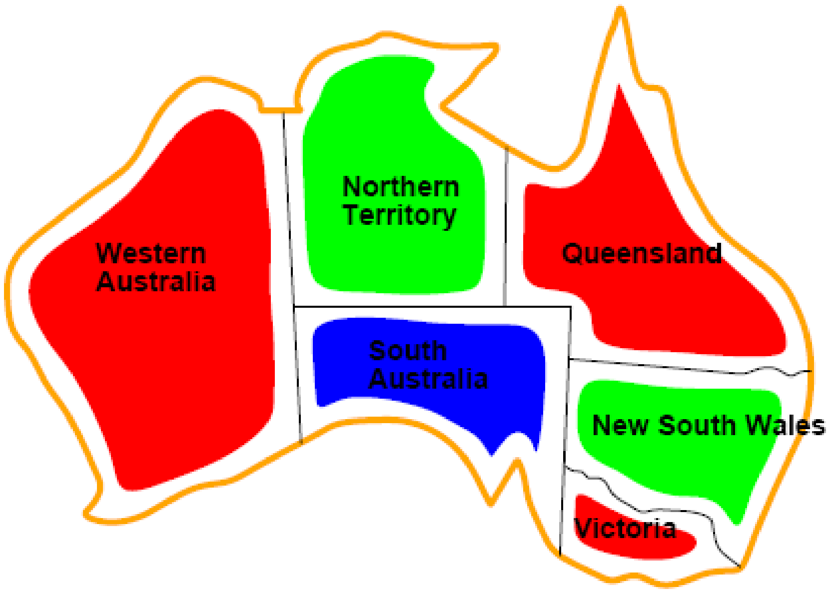
Introductive example #2

- Problem: colouring map of Australia
- Variables
  - WA, NT, Q, NSW, V , SA
- Domain
  - D_i = {red, green, blue}
- Constraint
  - Neighbor regions must have different colors
    - Color(WA) ≠ Color(NT)
    - Color(WA) ≠ Color(SA)
    - Color(NT) ≠ Color(SA)
    - …



Introductive example #2

- Solution is an assignment of variables satisfying all constraints, *e.g.* (if 'WA' stands for Color(WA)):
  - WA=red, and
  - NT=green, and
  - Q=red, and
  - NSW=green, and
  - V=red, and
  - SA=blue
- Other solutions exist



Constraint Satisfaction Problems

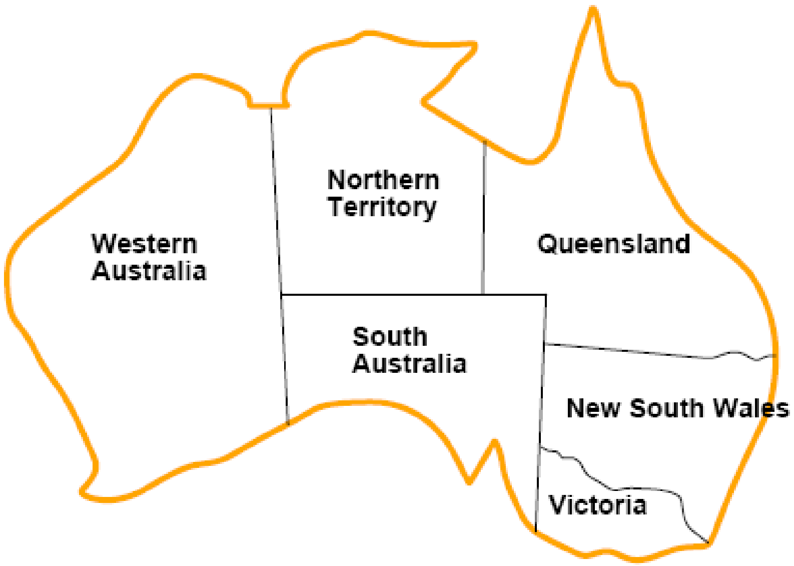
Definitions

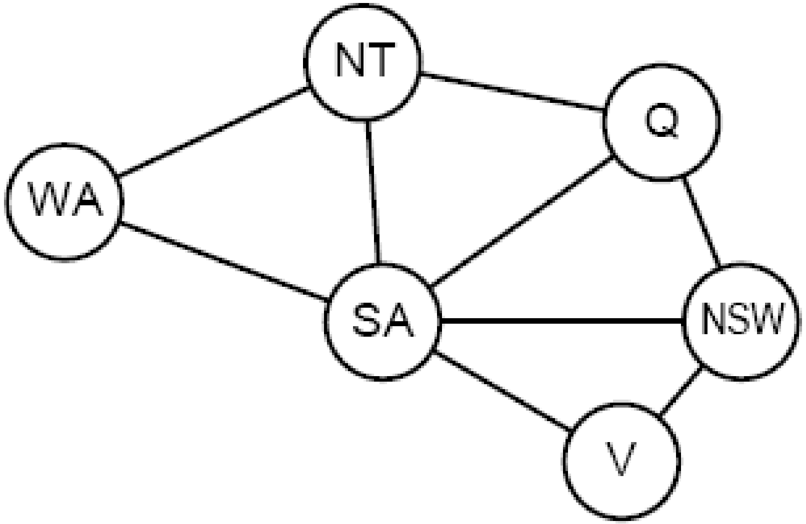
Different types of Constraints

- Unary (single-variable) constraints
  - *e.g.* SA ≠ green
- Binary constraints
  - *e.g.* SA ≠ WA
- Higher-order (*aka* **global** or multi-variable) constraints
  - Relate at least 3 variables, *e.g.*
    - Y is between X and Z, is a ternary constraint between(X, Y, Z)
    - Alldiff (in Sudoku rows and columns for instance)
- Soft constraints:
  - **Priority**, *e.g.*, red better than green
  - **Cost function** over variables

Constraint Graph

- In CSP, the constraints can be expressed using a graph:
  - Constraint graph
    - Node is variable
    - Edge (link) is constraint



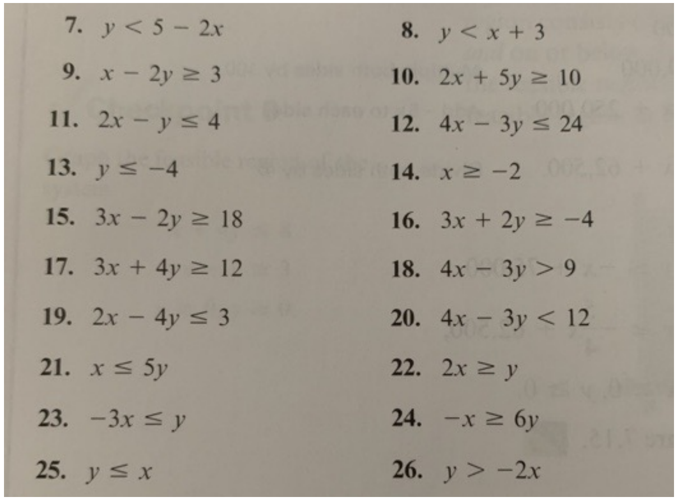


Types of variables

- Discrete variables can result in finite or infinite domains
  - Finite domain, *e.g.*, 8-queen and map coloring problems
  - Infinite domain, *e.g.* with integers or strings
    - With infinite domains, it is not possible to describe constraints by enumerating all allowed combinations of values (infinite)
    - Instead, a constraint language must be used
      - *E.g.* in a factory where task 2 must be performed at least d1 mn after Task1 (*e.g.* d1=time for paint to dry) *:*
        - Task1 + d1 ≤ Task2
    - Linear constraints (in which each variable appears only in linear form, as above) are solvable on integer variables
    - So far, there exists **no algorithm** for solving general non-linear constraint CSPs

Types of variables

- ❑ Continuous variables
  - CSPs with **continuous domains** are common in the real world, and widely studied in the field of operations research
    - *e.g.* start/end time of observing the universe using Hubble telescope
  - Linear constraints are solvable using Linear Programming
    - Constraints must be linear equalities / inequalities
    - About Linear Programming: https://byjus.com/maths/linear-programming/

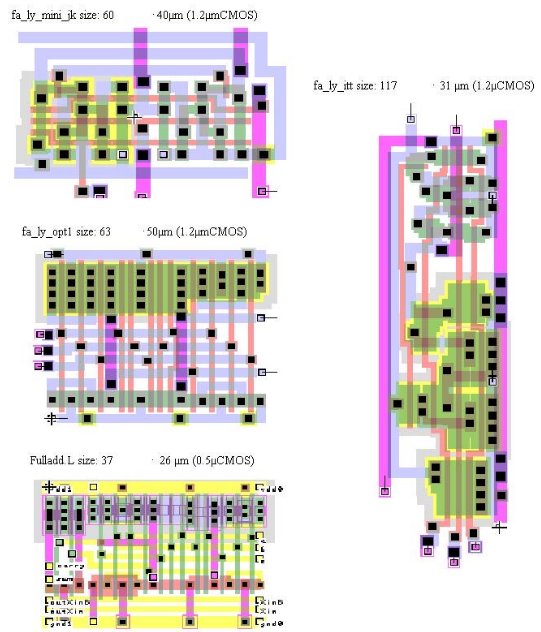


7. $y < 5 - 2x$
8. $y < x + 3$
9. $x - 2y \geq 3$
10. $2x + 5y \geq 10$
11. $2x - y \leq 4$
12. $4x - 3y \leq 24$
13. $y \leq -4$
14. $x \geq -2$
15. $3x - 2y \geq 18$
16. $3x + 2y \geq -4$
17. $3x + 4y \geq 12$
18. $4x - 3y > 9$
19. $2x - 4y \leq 3$
20. $4x - 3y < 12$
21. $x \leq 5y$
22. $2x \geq y$
23. $-3x \leq y$
24. $-x \geq 6y$
25. $y \leq x$
26. $y > -2x$

  - CSPs with different types of constraints / objective functions have also been studied, *e.g.* quadratic programming

Examples of Real-World CSP problems

- Assignment
  - *E.g.*, who teaches which class
- Scheduling
  - *E.g.*, when and where the class takes place
- Hardware design (*e.g.* VLSI layout)
- Transport scheduling
- Manufacture scheduling



Solving CSPs by Standard Search

- State
  - Defined by the values assigned so far
- Initial state
  - The empty assignment
- Successor function
  - Assign a value to an unassigned variable that **does not conflict** with current assignment + constraints
    - Fail if no “legal” assignment
- Goal test
  - All variables are assigned, and there is no conflict

Solving CSPs by Standard Search

- Characteristics of CSP problems
  - Every solution appears at depth *n* (with *n* the # of variables)
    - Use depth-first search
  - Path is irrelevant
    - It does not matter if NA was colored before NT, or NT before NA…
    - **So, local search** algorithms (hill climbing, simulated annealing…) can also be used, on top of other "AI-search" algorithms (*cf.* Chapter3 - part3)
  - But, the number of leaves is $n!d^n$ (with *d* the domain size)
    - Huge branching factor!!!

Constraint Satisfaction Problems

Backtracking search

Why is standard search not adapted to CSP?

- Standard search is extremely inefficient for CSP problems
  - Because they don't take advantage of the fact that CSP solutions are
    - A set of actions on separate variables (variable assignments)
      - Standard search algorithms use assignment of the whole set of variables at each step, instead of single-variable assignment
    - Where variable assignments are commutative
      - *E..g,* it does not matter is NA was colored before NT, or the other way around…

-> Backtracking search instead of standard search

Backtracking Search

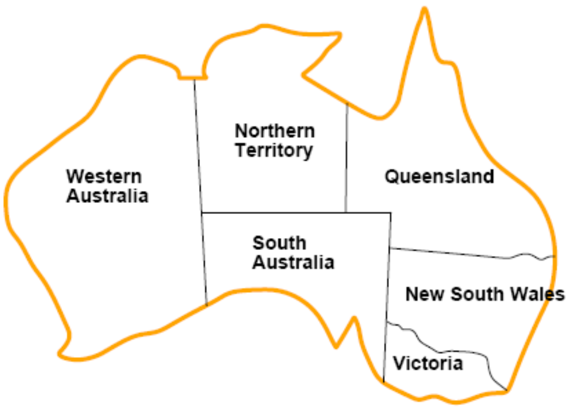
- Variable assignments are commutative, e.g.
  - {WA=red, NT =green}
  - {NT =green, WA=red}
- Single-variable assignment
  - Only consider one variable at each node
  - d^n leaves
- Backtracking search =
  - Depth-first search + Single-variable assignment
- Backtracking search is the **basic, uninformed** algorithm for CSPs
  - Can solve n-Queen with n = 25

Backtracking Search Algorithm

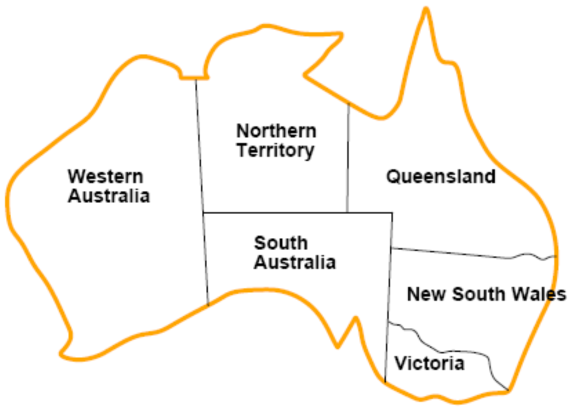
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    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) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

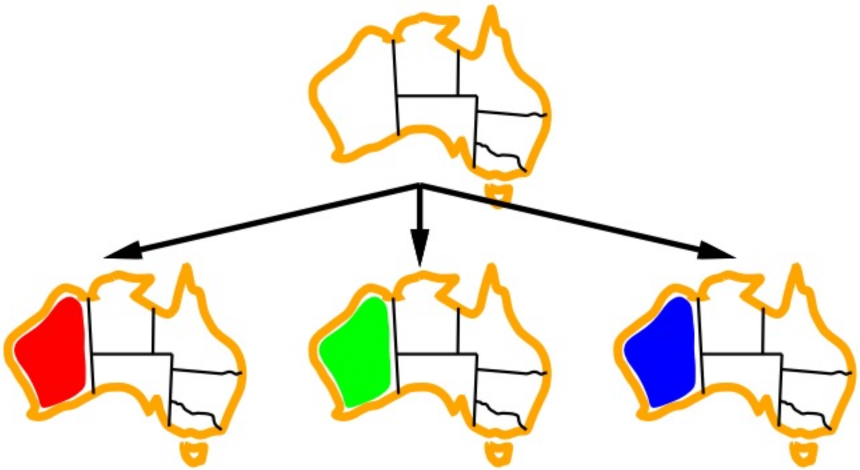
Called "**backtracking"** search because it's a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign

Backtracking Search Algorithm

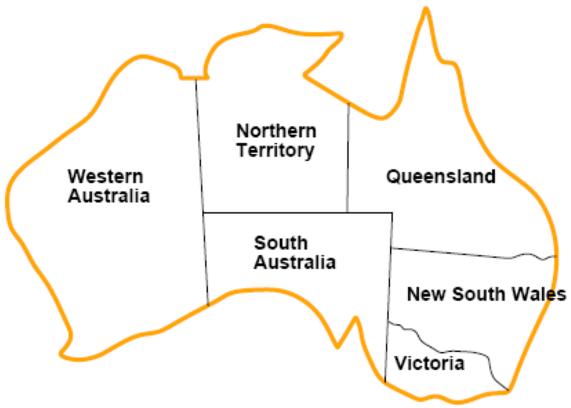


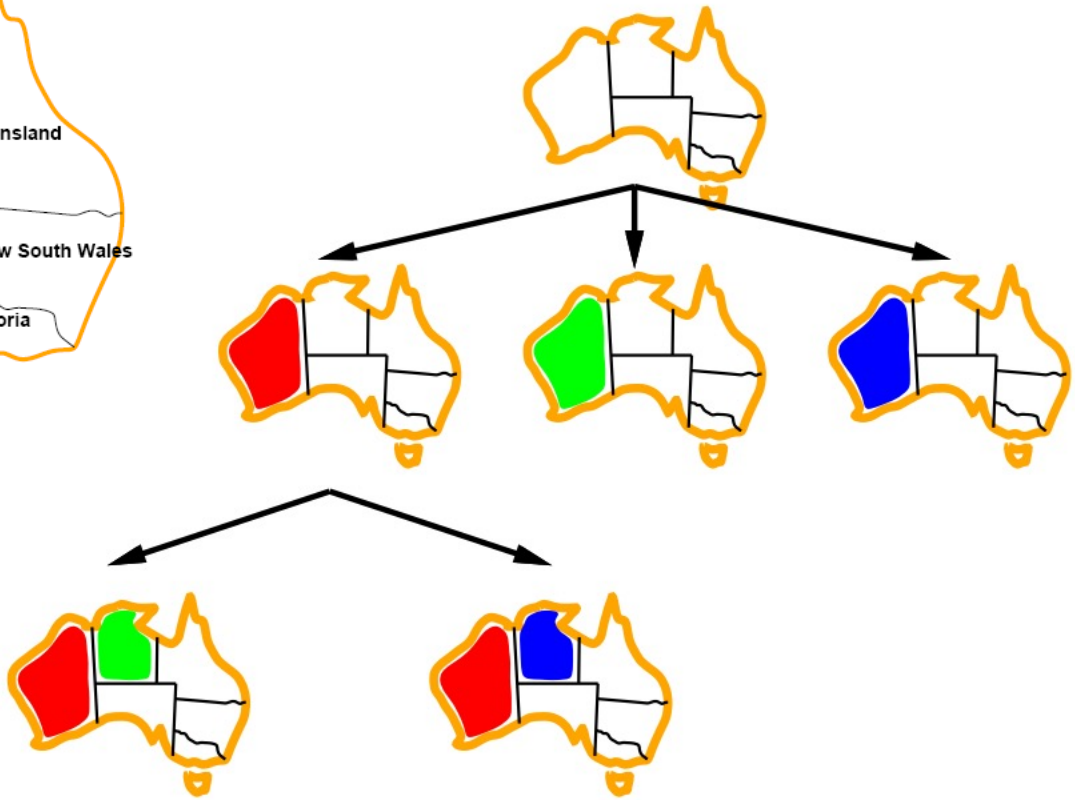
Backtracking Search Algorithm



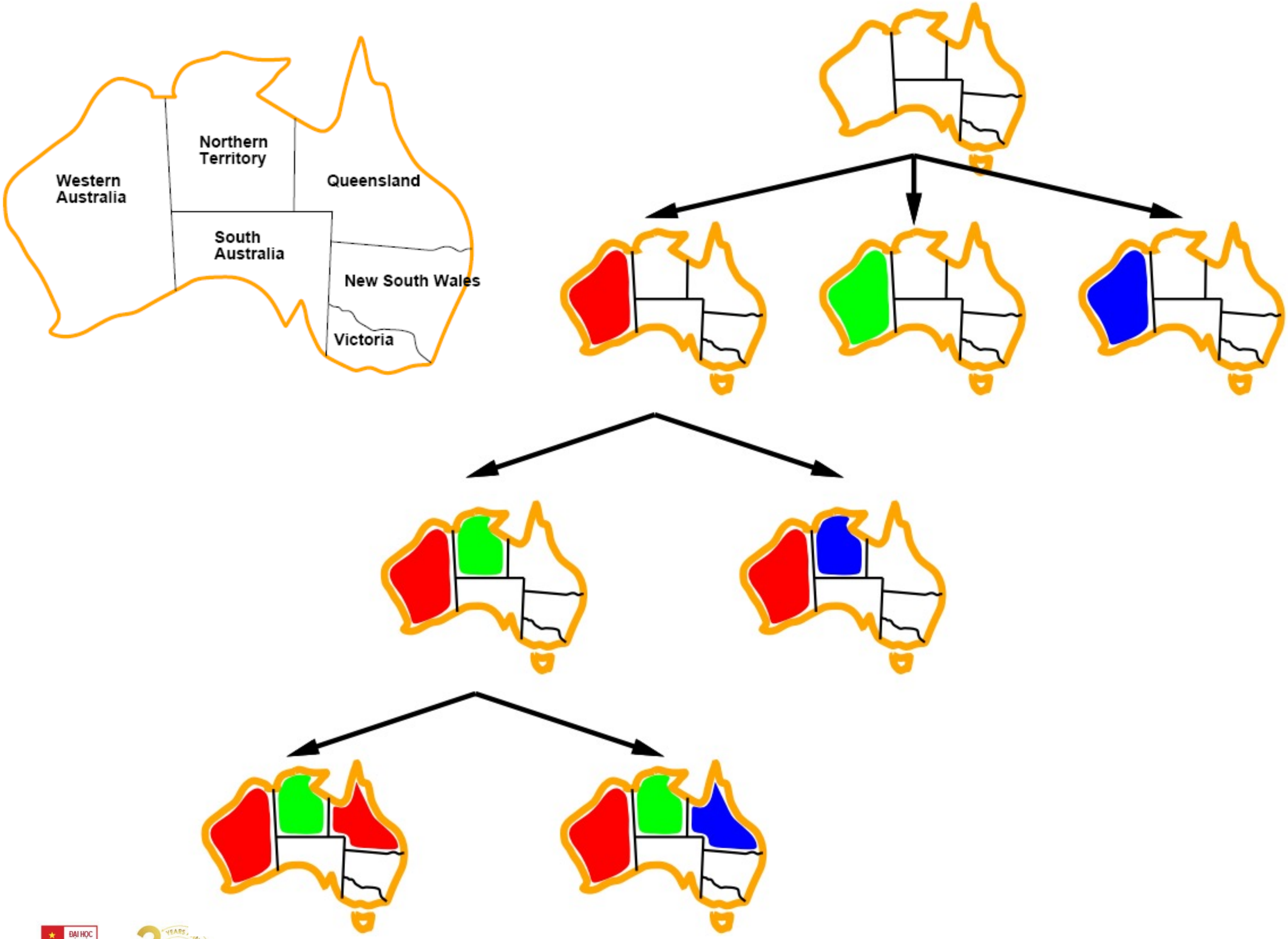


Backtracking Search Algorithm

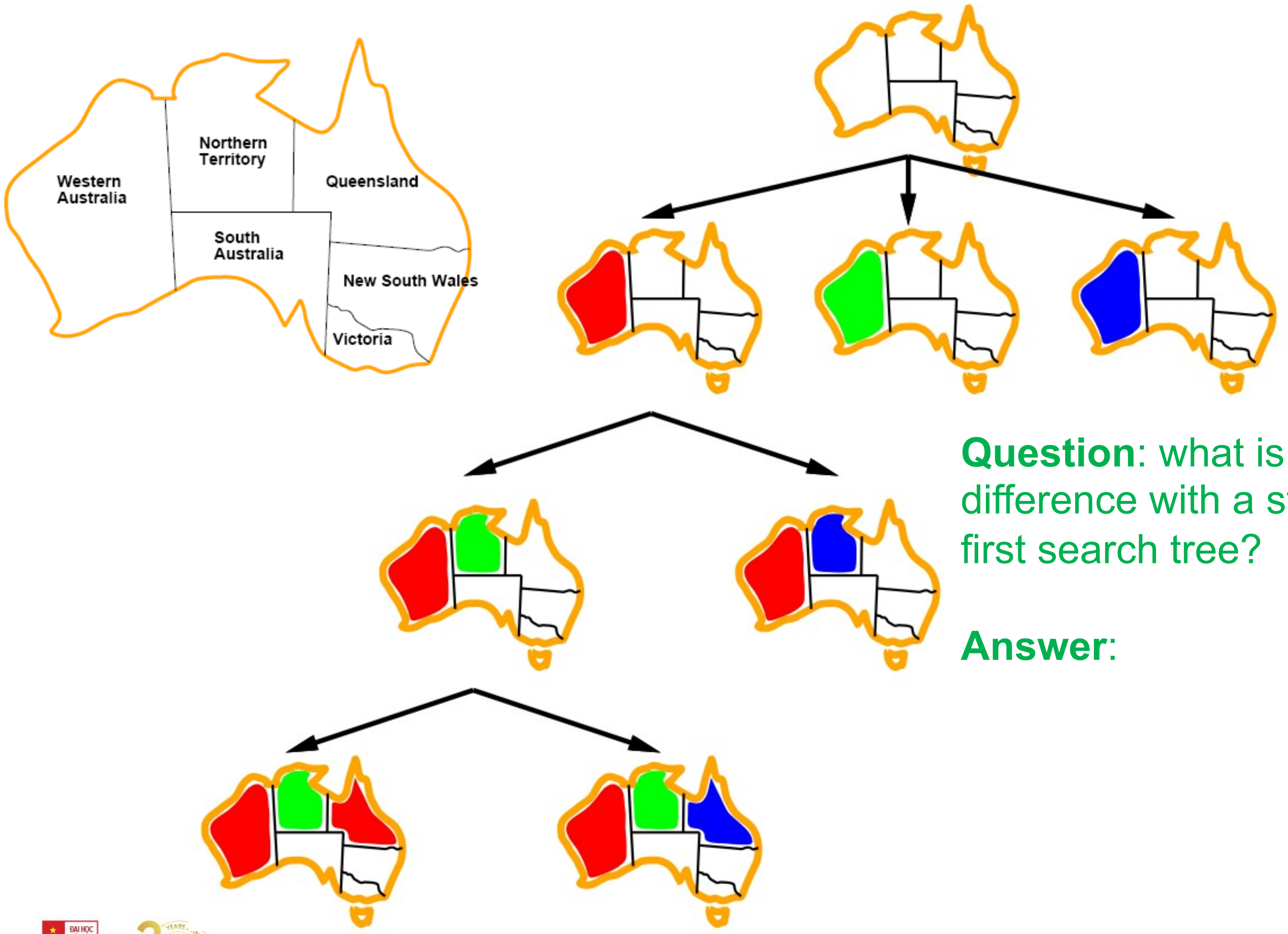




Backtracking Search Algorithm



Backtracking Search Algorithm



Question: what is the main difference with a standard depth-first search tree?

Answer:

Improving Backtracking Search

1. Which variable should be assigned next?
   - Function SELECT-UNASSIGNED-VARIABLE
2. In what order should its values be examined for assignment?
   - Function ORDER-DOMAIN-VALUE
3. Can we detect inevitable failure early?
4. Can we take advantage of problem structure?

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)
function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
   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) do
      if value is consistent with assignment given CONSTRAINTS[csp] then
         add {var = value} to assignment
         result ← RECURSIVE-BACKTRACKING(assignment, csp)
         if result ≠ failure then return result
         remove {var = value} from assignment
   return failure
```

1 - Choosing the next variable to assign

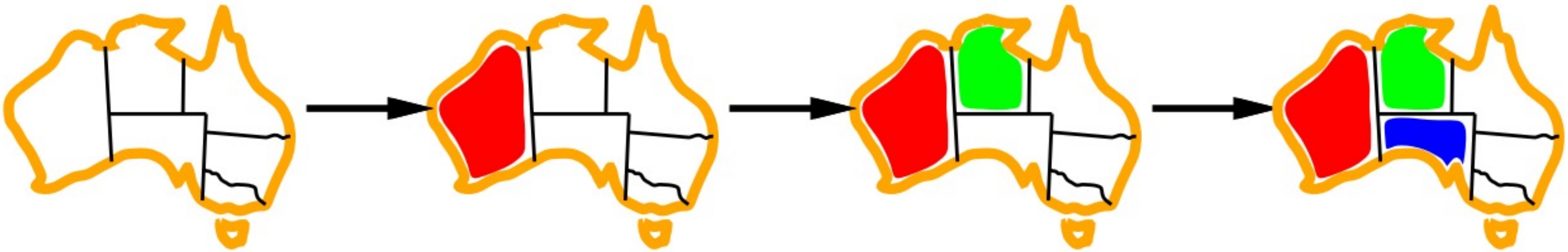
- Minimum remaining values (MRV)
  - Choose the variable with the fewest legal values
  - Idea: if a failure must come, let it come ASAP so that we can move to the next branch
  - The MRV heuristic performs better than a random or static ordering, up to a factor of 1,000 (depending on the problem)
  - But, the MRV heuristic can't help in choosing the 1^{st} region to color in Australia (initially, every region has 3 legal colors)

- -> Degree heuristic
  - Choose the variable with the most constraints on other remaining variables
  - Idea: reduce the branching factor on future choices

1 - Choosing the next variable to assign

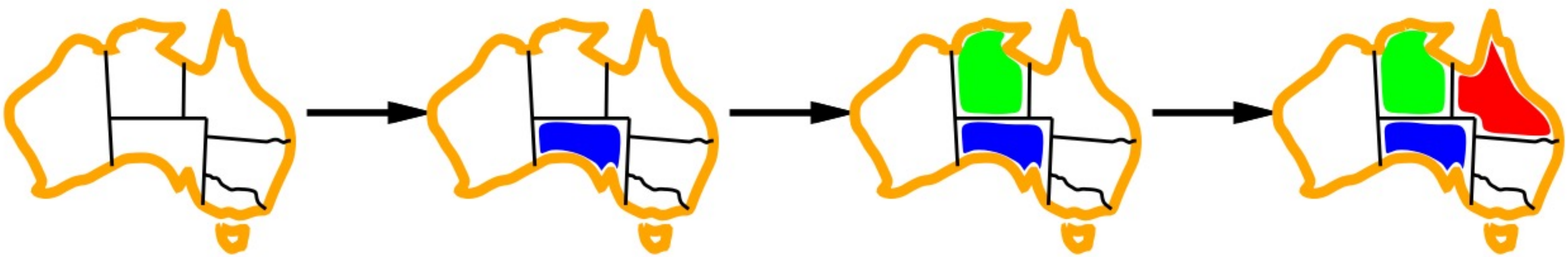
Minimum remaining values (MRV):

choose the variable with the fewest legal values



Degree heuristic:

choose the variable with the most constraints on remaining vars



Latter ofter used as a tie-breaker for former

[From http://homepage.cs.uiowa.edu/~tinelli/classes/4420/Spring17/notes/6-constraint-satisfaction.pdf]

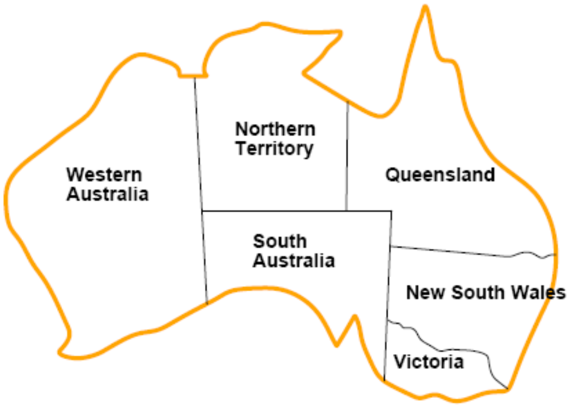
2 – Ordering the values to examine

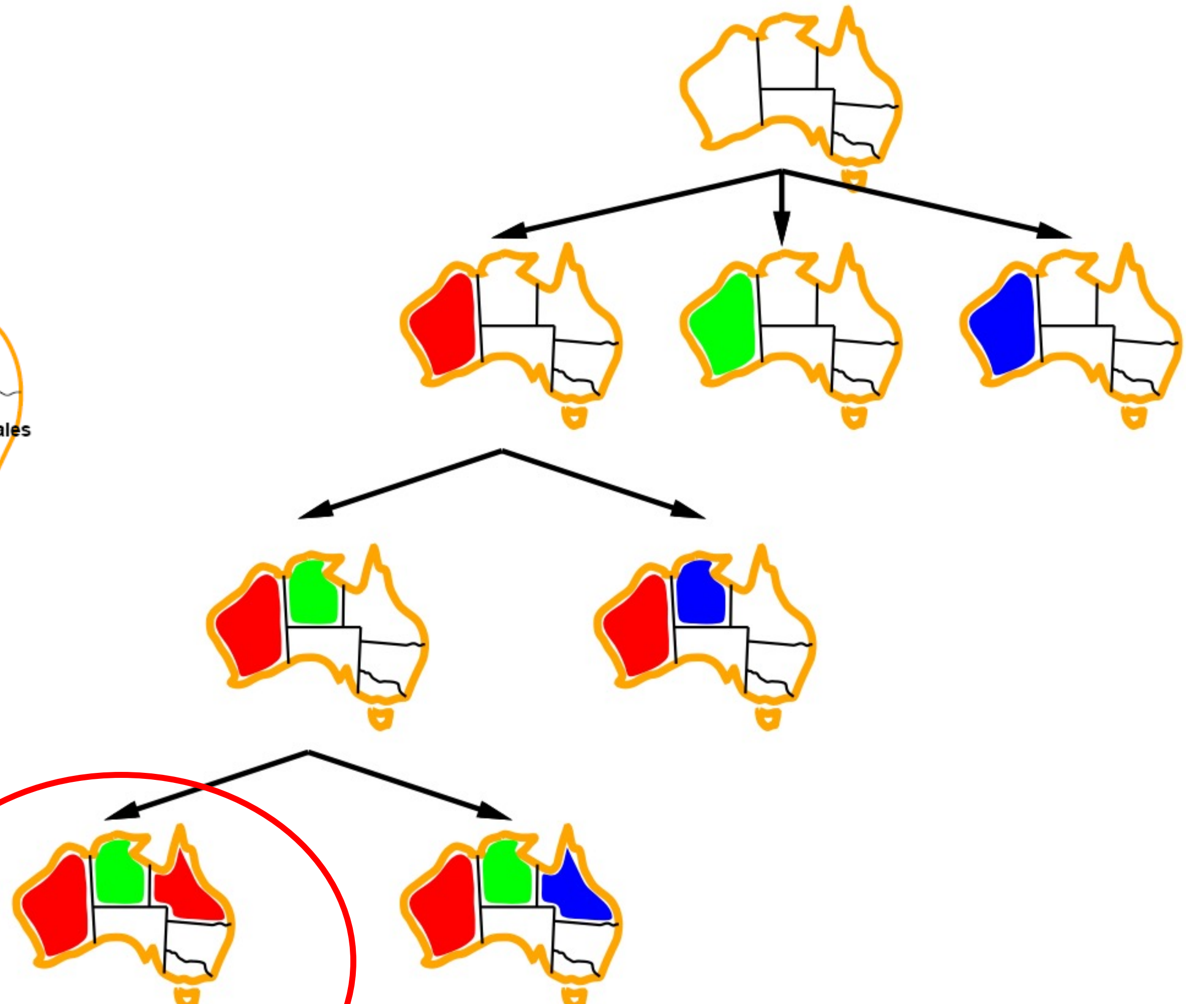
- **N.B.** Ordering the values to examine only has an interest if we are looking for **<u>any</u>** solution
  - **<u>Not</u>** if we want to list all possible solutions!

- Least constraining value (LCV)
  - Choose the least constraining value
    - the one that “forbids” the fewest values for the remaining variables

2 – Ordering the values to examine

- LCV: example





LCV, because 1 > 0*

Here we look only at WA because other states are constrained equally by the two possible colors for Queensland

Leaves 1 possible value for SA (blue)

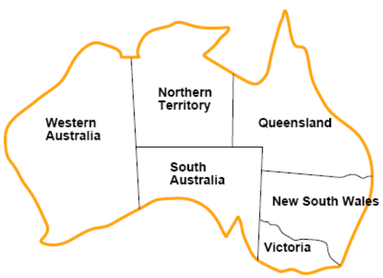
Leaves 0 possible value for SA

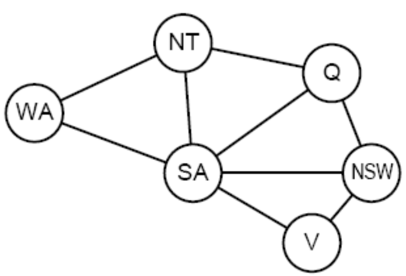
2 – Ordering the values to examine

- **Note**: Using MRV + degree heuristic + LCV, one can solve the 1000-queen problem (with a BIG, 1000x1000 board ☺)!
- But, we can go even further, by detecting early any future failure!
  - -> forward checking

3 - Can we detect inevitable failure early?

- **Idea** of forward checking:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any unassigned variable has no legal value
- **Step 0** of « simple » backtracking search with forward checking

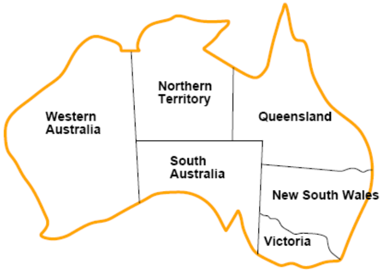


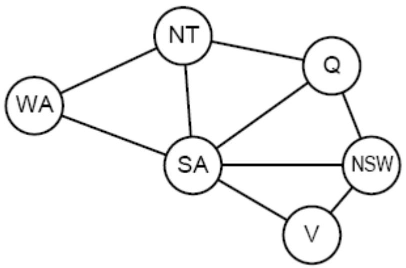


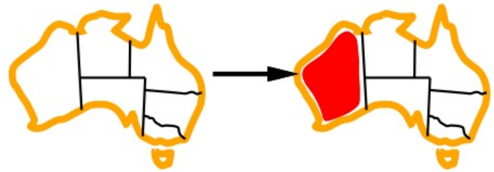
- All unassigned variables still have legal values -> continue

3 - Can we detect inevitable failure early?

- **Step 1** of « simple » backtracking search with forward checking



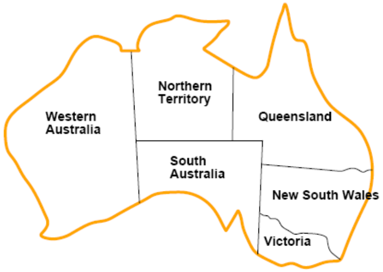


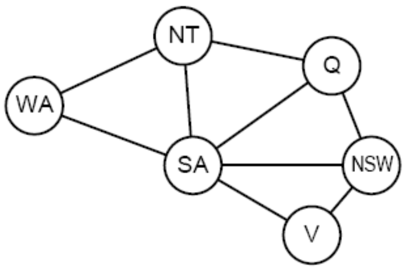


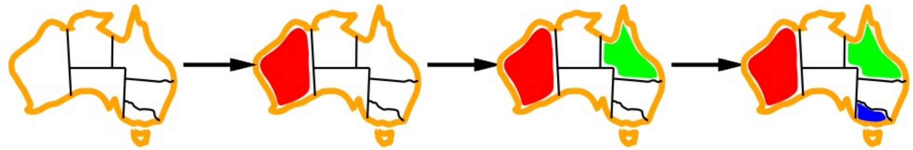
- All unassigned variables still have legal values -> continue

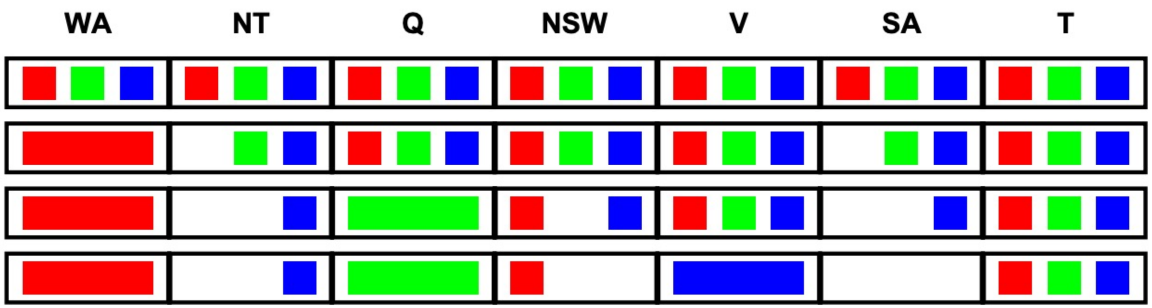
3 - Can we detect inevitable failure early?

- **Step 3** of « simple » backtracking search with forward checking







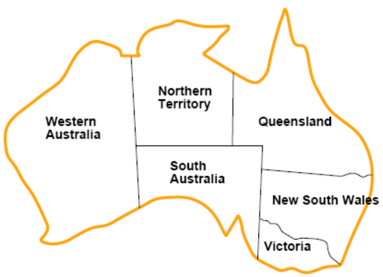


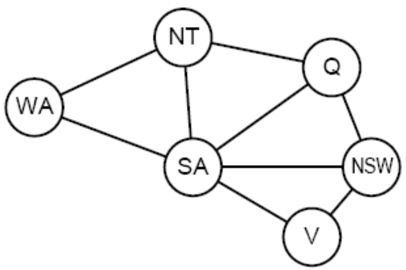
- With this configuration, SA does not have any possible value left -> prune search tree there (stop exploration of this branch)

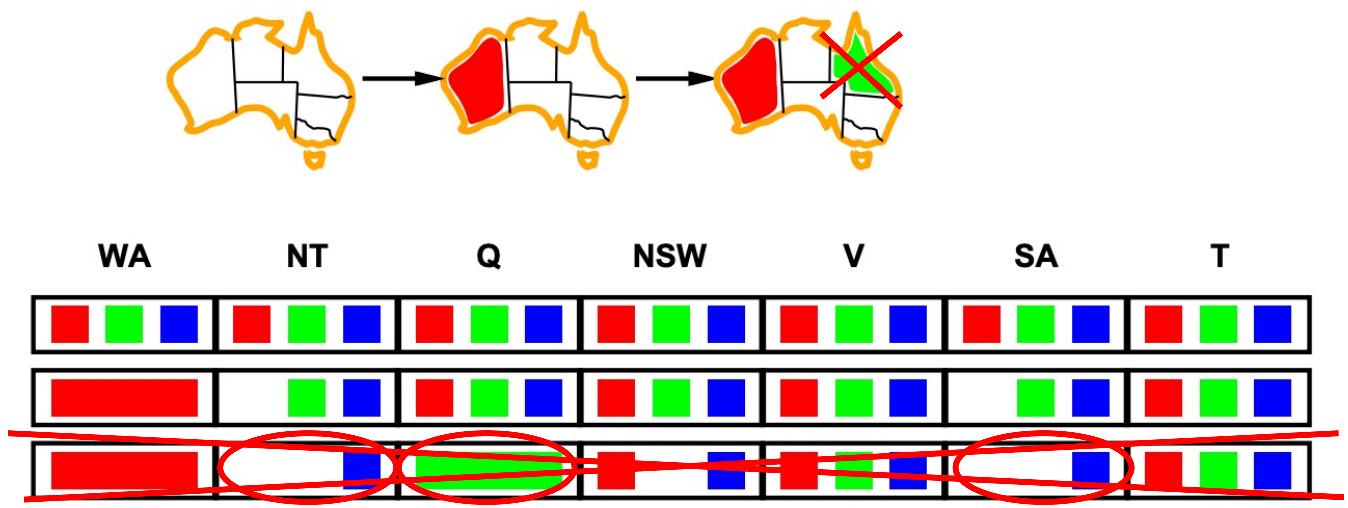
- This algorithm is called **forward checking**, because we did not yet choose which variable we will assign next, but we can detect that in any case, this will lead to failure
  - We're looking forward in the tree before devoping it -> forward checking

3 - Can we detect inevitable failure early?

- But, forward checking is not the best for detecting failure early
  - **Example:** at step 2 of «simple» backtracking search with forward checking







- In forward checking, all unassigned variables still have legal values -> continue
- **But**, inevitable failure could be detected here already:
  - Given that WA is red (step 1), if Q is green (step 2)…
    - then NT must be blue (given the constraints)
    - then SA must be blue (given the constraints)
    - but NT and SA are neighbours, so they cannot be BOTH blue!
  - **So**, Q cannot be green!

3 - Can we detect inevitable failure early?

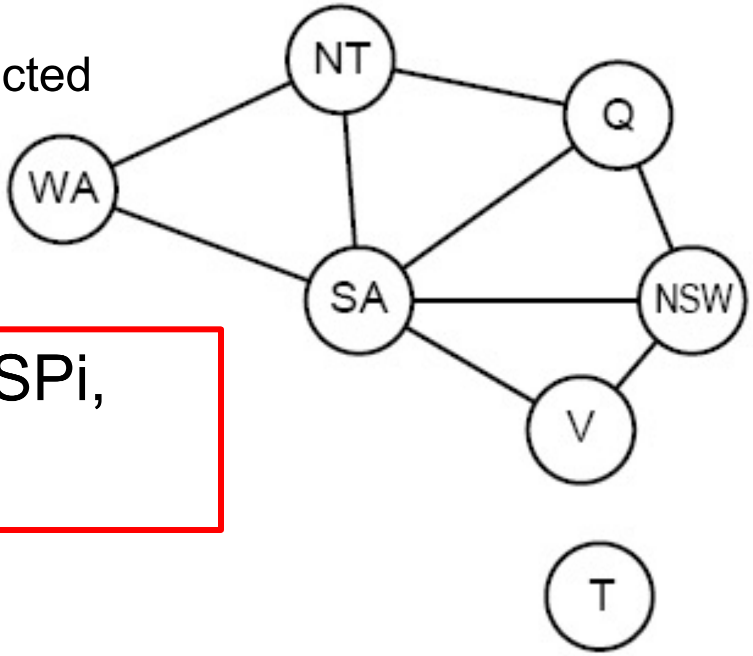
- Limitation of forward checking:
  - Does not verify, at each step, that each arc is **consistent**
    - X -> Y is consistent **iff** for each value x of X there is some allowed value y for Y

  -> **arc consistency** algorithm (out of the scope of this course)

    https://www.cs.ubc.ca/~kevinlb/teaching/cs322%20-%202009-10/Lectures/CSP3.pdf

4 - Can we take advantage of problem structure?

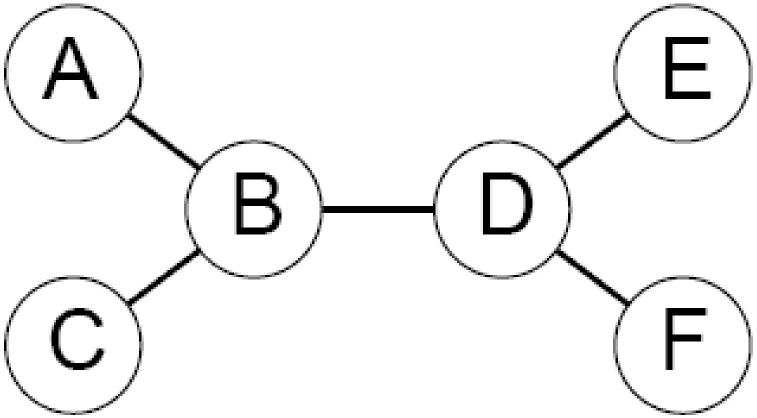
- **Special case #1**: Independent subproblems
  - *E.g.* assume we have a new region T (Tasmania) to consider
  - The problem of T and the rest are two independent sub-problems CSP_i
    - Each sub- problem is a set of connected component in the constraint graph



If assignment S_i is a solution of CSPi, then U_iS_i is a solution of U_iCSP_i.

4 - Can we take advantage of problem structure?

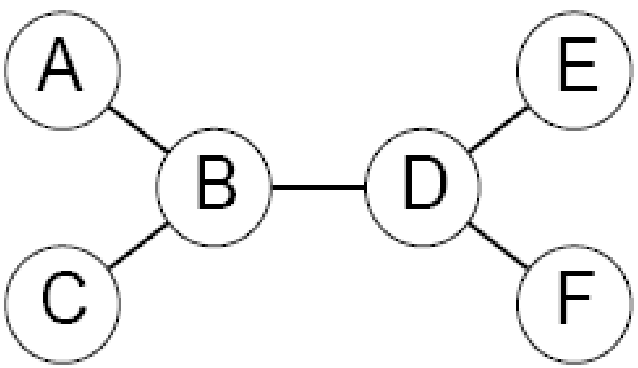
- **Special case #2**: tree-structured problem
  - Any two variables are connected by only one **path**
  - Example**:**

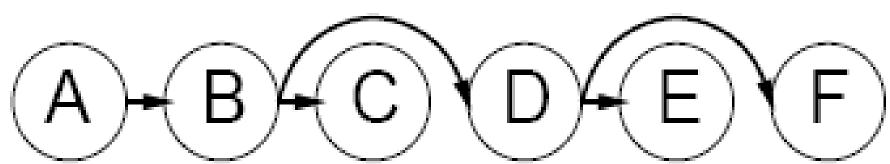


- **Theorem**
  - If the constraint graph has no loop then CSP can be solved in $O(nd^2)$ time

4 - Can we take advantage of problem structure?

- Algorithm for tree-structured problems
  - To solve a tree-structured CSP, create a **topological sort** (several possible in general)
  - Then, specific algorithms can solve it in linear time
    - For more details and the proof, check the reference book.





Constraint Satisfaction Problems

Summary

Summary

- CSPs are a special kind of problem:
  - states defined by values of a fixed set of variables
  - goal test defined by constraints on variable values (after all variables have been assigned a value)
- Backtracking = depth-first search with 1 variable assigned per node
- Variable ordering and value selection heuristics help significantly
  - Variable ordering: usually MRV + degree heuristic in case of ties
  - Values ordering: usually, LCV
- Forward checking prevents assignments that guarantee later failure
  - Arc consistency does additional work to constrain values and detect inconsistencies as early as possible
    - Out of the scope of this course, but very interesting -> please have a look
- In some special cases, the CSP problem structure can be taken advantage of
  - Independent sub-problems can be solved jointly, and then their solutions joined
  - Tree-structured CSPs can be solved in linear time thanks to a dedicated algorithm

Constraint Satisfaction Problems

Exercise / homework

Exercice: cryptarithmetic problem

- Problem:
  - Each letter corresponds to a digit 0..9
  - Each letter corresponds to a different digit
  - F cannot be 0

Variables: $F,\ T,\ U,\ W,\ R,\ O,\ X_1,\ X_2,\ X_3$

Domain: $\{0,1,2,3,4,5,6,7,8,9\}$

Constraints: $alldiff(F,T,U,W,R,O)$

Tip:
You’ll need 3 extra variables
to solve this problem (if the sum of 2 letters >= 10)

- Solve this problem by combining:
  - Constraint Propagation
  - Minimum Remaining Values
  - Least Constraining Values

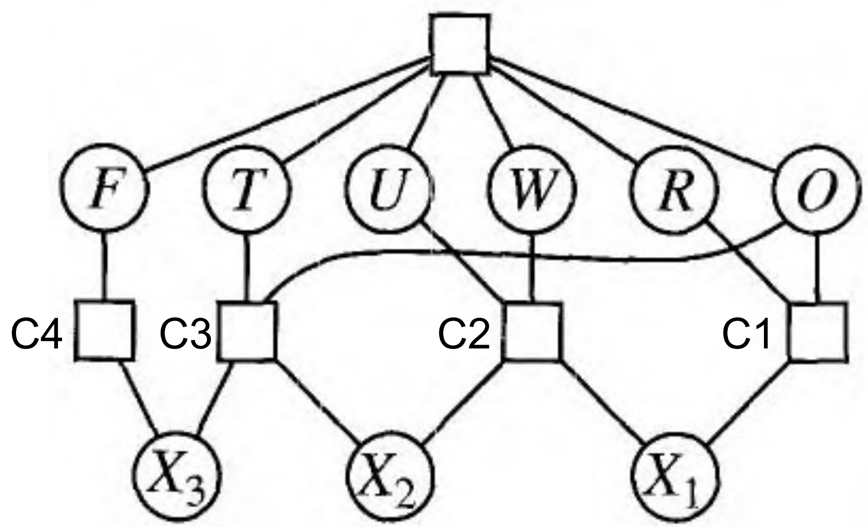
- Note: there are several solutions to this problem

Exercice solution

- First step: define extra variables X1, X2, X3 and their constraints
  - Constraints and domains:
    - T ∈ {0,...,9}; W ∈ {0,...,9}; 0 ∈ {0,...,9};
      F ∈ {0,...,9};U ∈ {0,...,9}; R ∈ {0,...,9}
    - X1 ∈ {0,1}; X2 ∈ {0,1}; X3 ∈ {0,1}
    - C1: 0+0=R+10*X1
    - C2: X1+W+W=U+10*X2
    - C3: X2+T+T=O+10*X3
    - C4: X3=F
    - C5: F!=0
    - C6: *Alldiff*(T,W,O,F,U,R)

```
   TWO
+  TWO
------
  FOUR
```

- Second step: build the constraint hypergraph



Homework

- Find at least 2 other solutions to this problem, by using the same strategies

Chapter 3 – part 5

Questions

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you
for your
attention!