Artificial Intelligence II

2013/2014 - Prof: Daniele Nardi, Joachim Hertzberg

Exercitation 4 - Roberto Capobianco

Planning: HTN, Notes on CSPs

Hierarchical Task Networks (Recap)

- Planning goals specified in terms of tasks:
  - Primitive tasks;
  - Non-primitive tasks (decomposed by methods in sub-tasks);
- Plans are task networks;
- Linear vs. Nonlinear decomposition;
- Planning domain: methods, tasks, facts;
- Planning problem: domain, initial situation, set of goal tasks;
- Total-order planning: the steps are scheduled in the same order as they are executed later.

SHOP: Simple Hierarchical Ordered Planner

- The plan is expanded one action at a time in the order in which it will be executed:
  - Conflict reduction;
  - Simplification;
- The planner always knows the full state of the world (i.e., all the consequences of each action until that point).

SHOP: Simple Hierarchical Ordered Planner

- SHOP uses first order logics;
- A state is a set of atoms (facts);
- A task has the general form $(s\ t_1\ t_2\ t_3\ ...\ t_n)$;
  - s is the task name;
  - t_i are the task arguments;
  - Primitive tasks begin with !;
  - Non-primitive tasks don't have special characters;

SHOP: Operators

(:operator h D A)

- An operator is a expression where:
  - h (the head) is a primitive task
  - D represents the deletions;
  - A represents the additions;
  - D and A are sets of atoms containing no variable symbols other than those in h;
- It can be accomplished by modifying the current state of the world to remove every atom in D and add every atom in A.

SHOP: Operators (Example)

- An operator to put a block on the table:

```
(:operator (!putdown ?block)
((holding ?block))
((ontable ?block) (handempty)))
```

SHOP: Methods

(:method h C T)

- An operator is a expression where:
  - h (the head) is a compund task
  - C is a conjunct representing the method's preconditions;
  - T is a task list and represents the method's tail;
- If the current state of the world satisfies C, the method can be accomplished by performing the tasks in T in the order given;
- A method can have multiple pairs of preconditions and tails, to be used in an "if-then-else" fashion. For example, (:method head pre1 tail1 pre2 tail2) says that the reduction of head is tail1 if pre1 is true, or tail2 if pre1 is false and pre2 is true.

SHOP: Methods (Example)

- If y is already clear, do nothing;
- If another block x is on y, make x clear and move x to the table.

```
(:method (makeclear ?y) ((clear ?y)) nil
(:method (makeclear ?y)
((on ?x ?y))
((makeclear ?x)
(!unstack ?x ?y) (!putdown ?x))
```

SHOP: Axioms

- The axioms are generalized versions of Horn clauses:

(: - head tail)

- head is true if tail is true;
- The tail of the clause may contain anything that may appear in the precondition of an operator or method;
- Axioms can have multiple tails: head is true if tail1 is true, or if tail1 is false but tail2 is true, etc.

SHOP: Axioms (Example - 1)

- A plane has enough fuel to reach ?destination if the following conditions are satisfied:
  - The distance to travel is ?dist;
  - The fuel level is ?fuel-level;
  - The burn rate is ?rate;
  - ?fuel-level is not less than the product of ?rate and ?distance.

SHOP: Axioms (Example - 2)

```
(:-
(enough-fuel ?plane ?current-position ?destination ?speed)
(and (distance ?current-position ?destination ?dist)
(fuel ?plane ?fuel-level)
(fuel-burn ?speed ?rate)
(eval (>= ?fuel-level (* ?rate ?dist)))))
```

NOTE: The last of these conditions is handled using an external function call: external function calls are useful, for example, to do numeric evaluations.

SHOP Example

- We want to travel from one location to another in a city.
- There are three possible modes of transportation: taxi, bus, and foot.
- Taxi travel involves hailing the taxi, riding to the destination, and paying the driver $1.50 plus $1.00 for each mile traveled.
- Bus travel involves hailing the bus, paying the driver $1.00, and riding to the destination.
- Foot travel just involves walking, but the maximum feasible walking distance depends on the weather.
- Different plans are possible depending on what the layout of the city is, where we start, where we want to go, how much money we have, and what the weather is like.

SHOP Example: Initial State & Task List

Initial State:

```
((at downtown)
 (weather-is good)
 (have-cash 12)
 (distance downtown park 2)
 (distance downtown uptown 8)
 (distance downtown suburb 12)
 (at-taxi-stand taxi1 downtown)
 (bus-route bus1 downtown park)
 (bus-route bus2 downtown uptown)
 (bus-route bus3 downtown suburb))
```

Task list:

```
((travel-to suburb))
```

SHOP Example: Axioms

- The taxi fare is $1.50 plus $1 for each mile traveled:

```
(: (have-taxi-fare ?distance)
        ((have-cash ?m)
        (eval (>= ?m (+ 1.5 ?distance)))))
```

- Walking distance is < 3 miles in good weather, and < 1 mile otherwise:

```
(: (walking-distance ?u ?v)
        ((weather-is good)
        (distance ?u ?v ?w)
        (eval (<= ?w 3)))
        ((distance ?u ?v ?w)
        (eval (<= ?w 1))))
```

SHOP Example: Operators

```
(:operator (!hail ?vehicle ?location)
    ()
    ((at ?vehicle ?location)))
(:operator (!wait-for ?bus ?location)
    ()
    ((at ?bus ?location)))
(:operator (!ride ?vehicle ?a ?b)
    ((at ?a) (at ?vehicle ?a))
    ((at ?b) (at ?vehicle ?b)))
(:operator (!set-cash ?old ?new)
    ((have-cash ?old))
    ((have-cash ?new))
(:operator (!walk ?here ?there)
    ((at ?here))
    ((at ?there))
```

SHOP Example: Methods

```
(:method (pay-driver ?fare)
        ((have-cash ?m) (eval (>= ?m ?fare)))
         ((!set-cash ?m, (- ?m ?fare))))
(:method (travel-to ?q)
        ((at ?p) (walking-distance ?p ?q))
        ((!walk ?p ?q)))
(:method (travel-to ?y)
        (:first (at ?x) (at-taxi-stand ?t ?x)
           (distance ?x ?y ?d) (have-taxi-fare ?d))
        ((!hail ?t ?x) (!ride ?t ?x ?y) (pay-driver (+ 1.50 ?d)))
        ((at ?x)(bus-route ?bus ?x ?y))
        ((!wait-for ?bus ?x) (pay-driver 1.00) (!ride ?bus ?x ?y)))
```

NOTES:

- :first is used to tell SHOP that it should only consider hailing the first taxi at the taxi stand, rather than hailing all of them;
- The third method specifies that we won't consider bus travel unless we don't have enough money for taxi travel.

SHOP2

- SHOP2 relaxes the total-order strategy (possibility for partial-order-planning);
- The order of execution is not defined as total-order for all subtasks;
- Same syntax and semantics as SHOP;
- SHOP chooses at the beginning of the first task from the task list / SHOP2 selects non-deterministically a task from the task list, which has no predecessor;

JSHOP2

- JSHOP2 is the Java implementation of SHOP2;
- Available for download at: http://www.cs.umd.edu/projects/shop/ (download the GUI version)
- JSHOP2 knows the current state of the world at each step of the planning process;
- It has large expressive power. For example, in the preconditions of operators and methods it can do mixed symbolic/numeric computations and execute calls to external programs.
- JSHOP2 incorporates many features from PDDL, e.g., support for quantifiers and conditional effects in methods and operators.
- JSHOP2 allows the combination of partially ordered tasks through the use of the :unordered keyword.

JSHOP2GUI: Obtaining a Plan (1)

- Compile the domain description into Java code:

  ```
  java JSHOP2.InternalDomain InputFileName
  ```

  - The result of this command will be a Java file (named after the domain name in the input domain description) that contains a class (with the same name) that implements the functionality of the input JSHOP2 domain;

- Compile the problem descriptions into Java code:

  ```
  java JSHOP2.InternalDomain -r InputFileName
  ```

  - The result of these commands will be a Java file (named after the problem name in the input problem description) that contains a class (with the same name) that implements the functionality of the input JSHOP2 planning problems;

- Compile and run the Java file produced in the previous step.

JSHOP2GUI: Obtaining a Plan (2)

- Create a file gui.java:

```
import JSHOP2.*;
import java.util.*;

public class gui {
        public static void main(String[] args) {
                problem.getPlans();
                new JSHOP2GUI();
        }
}
```

- Compile and run it. When this file is run, it will find the plan(s) for the input planning problem(s) and print it/them by clicking on the "Run" button.

Part II – Notes on CSPs

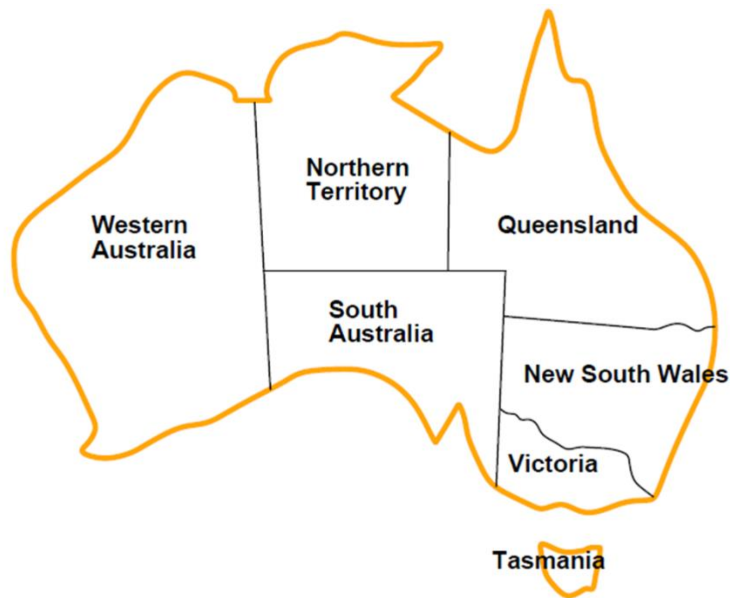
(useful for understanding Temporal Reasoning)

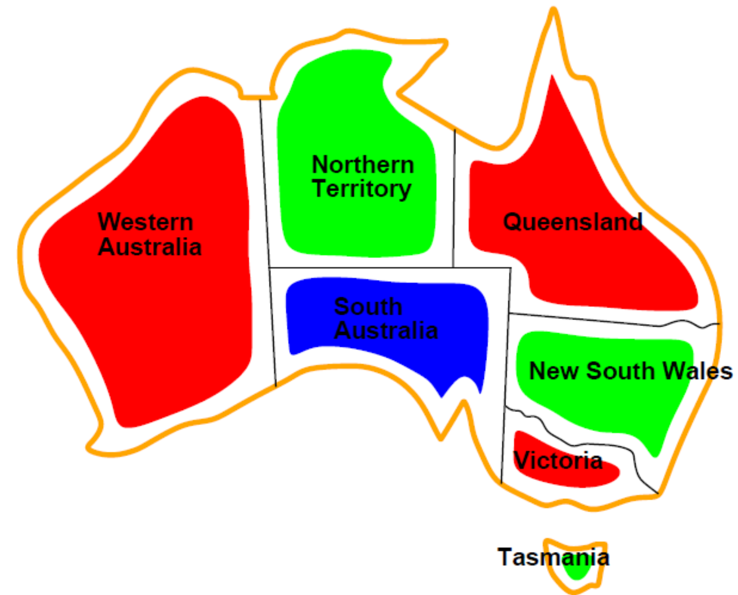
Constraint satisfaction problems (CSPs)

- A state is defined by variables X_i with values from domain D_i
- Assignments of variables to values are:
  - Consistent (or legal), if it doesn't violate any constraint;
  - Complete, if all the variables are assigned;
- A solution is a complete and consistent assignment;
- A goal test is a set of constraints specifying allowable combinations of values for subsets of variables;

CSPs Example: Map Coloring

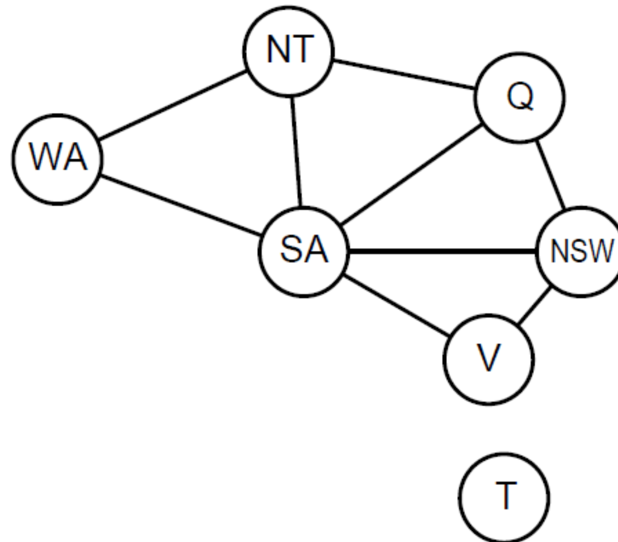
- Variables WA, NT, Q, NSW, V, SA, T;
- Domains Di = {red, green, blue};
- Constraints: adjacent regions must have different colors;





Constraint Graph

- General purpose CSP algorithms use the graph structure to speed up search (e.g., Tasmania is an independent subproblem);
- Constraint graph: nodes are variables, arcs show constraints.



Varieties of CSPs

- Discrete variables:
  - Finite domain (common);
  - Infinite domain (e.g, integers, strings): reduced to finite by putting an upper bound to values;
- Continuous variables:
  - Linear programming (Operation Research).

Varieties of Constraints

- Unary constraints, involving a single variable (e.g., SA ≠ green);
- Binary constraints, involving pair of variables (e.g., SA ≠ WA);
- Higher order constraints involving 3 or more variables;
- Preferences or soft constraints (e.g., red is better than green), usually represented by a cost for each variable assignment.

Standard Search Formulation

- Initial state: { };
- Successor function: assign a value to an unassigned variable that does not conflict with current assignment $\Rightarrow$ fail if no legal assignments (not fixable!);
- Goal test: the current assignment is complete;
- Path cost: a constant cost for every step;

- This is the same for all CSPs;
- Every solution appears at depth n with n variables $\Rightarrow$ use depth-first search
- Path is irrelevant, so can also use complete-state formulation
- $b=(n-\ell)d$ at depth ℓ, hence $n!d^n$ leaves.

Backtracking Search

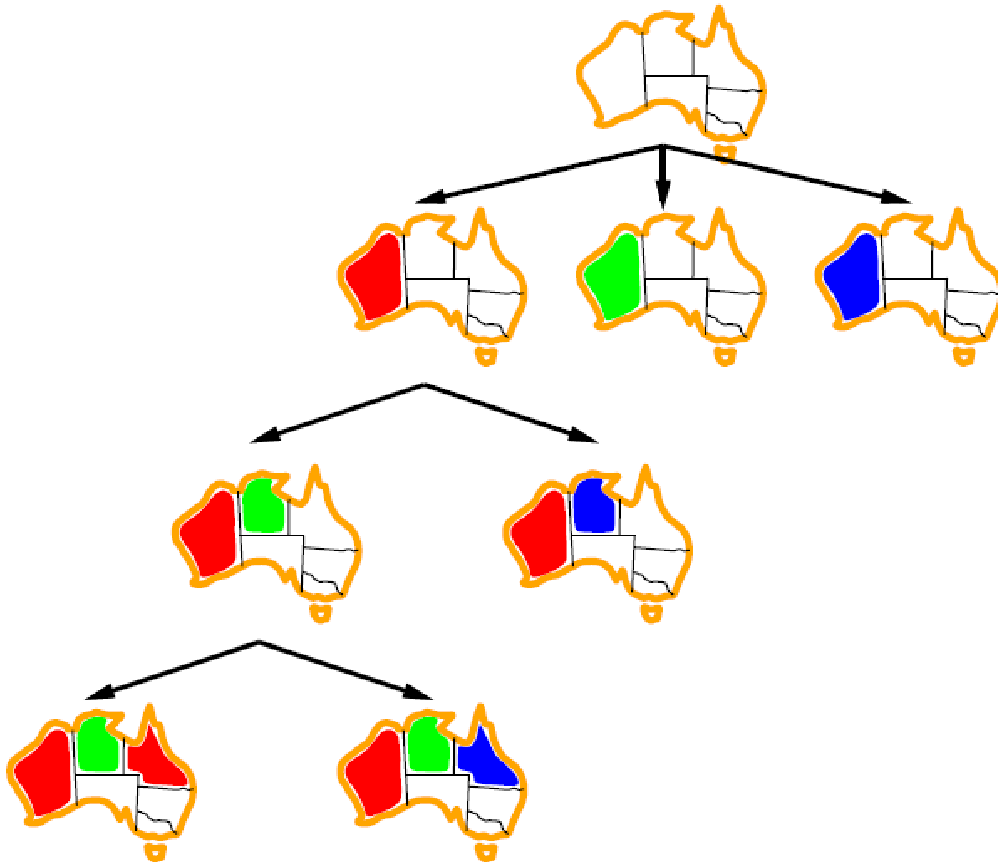
- Variable assignments are commutative, (e.g., [WA=red, NT =green] same as [NT =green, WA=red]);
- Only need to consider assignments to a single variable at each node ⇒ b=d and there are d^n leaves;
- Depth-first search for CSPs with single-variable assignments is called backtracking search;
- Backtracking search is the basic uninformed algorithm for CSPs;

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
```

Backtracking Search: Example



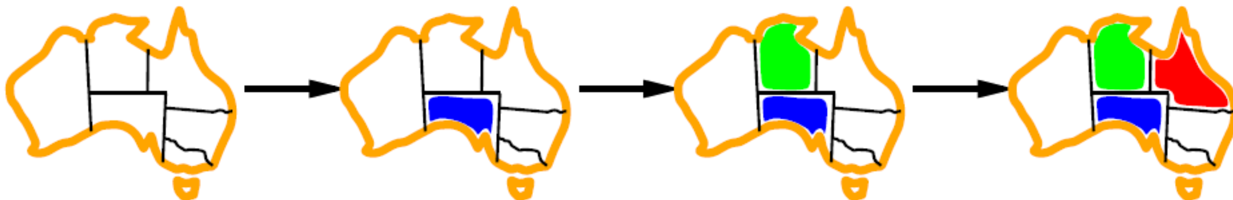
Improving Backtracking

- Backtracking can be affected by:
  - The order in the assignment of variables;
  - The order of the values tried;
  - The problem structure;
  - An early failure detection;

Variable Order

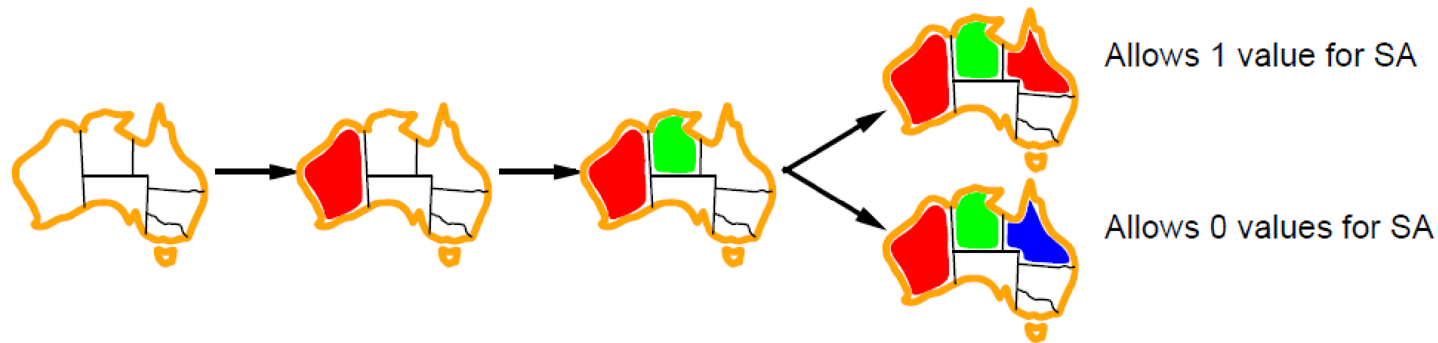
- Choose the variable with the fewest legal values (Minimum Remaining Values):

- Tie breaker: degree heuristic – choose the variable with the most constraints on remaining variables;



Value Order

- Given a variable, choose the least constraining value;
- Choose the value that rules out the fewest values in the remaining variables;



Early Detection (1)

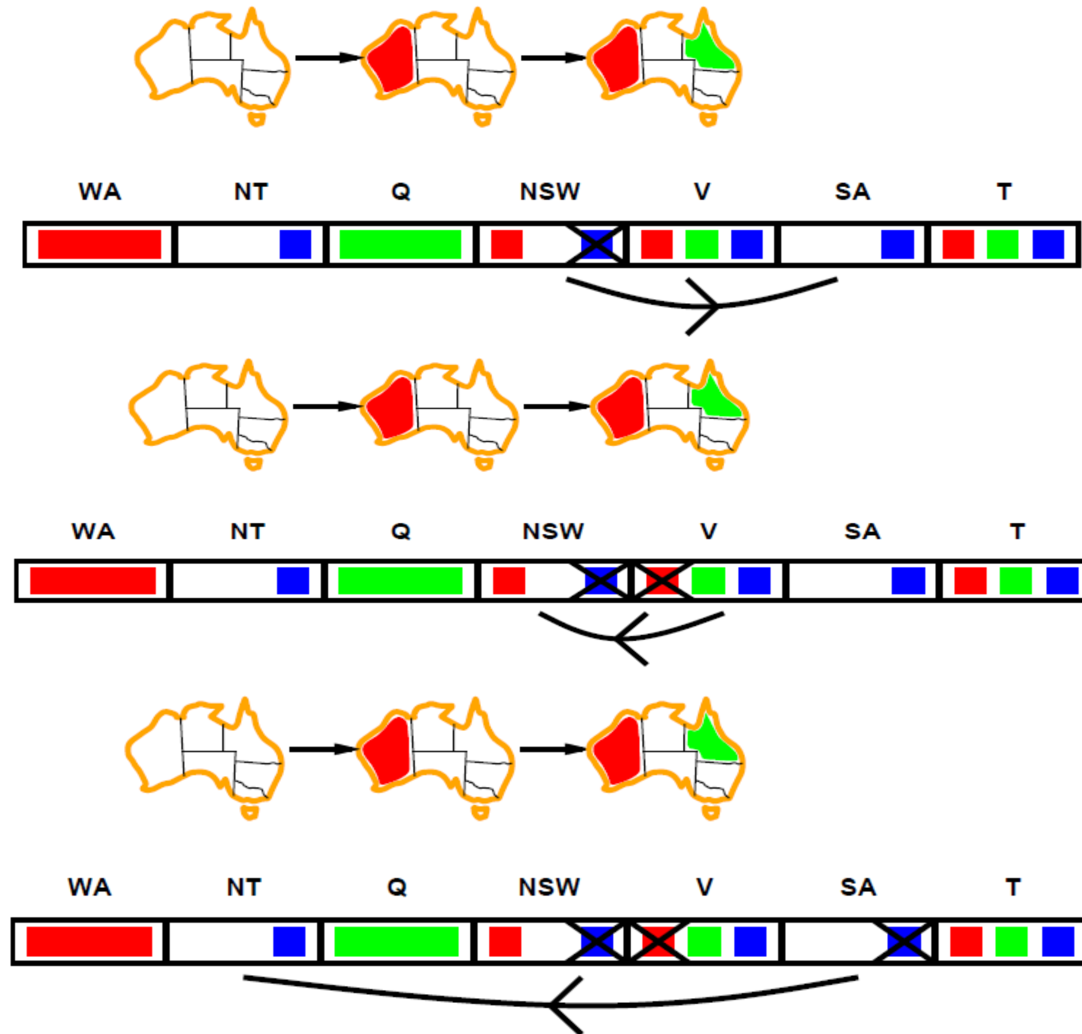
- Keep track of remaining legal values for unassigned variables;
- Terminate search when any variable has no legal value;

WA NT Q NSW V SA T

Early Detection (2)

- NT and SA cannot both be blue;
- Constraint propagation enforces constraints:
  - The simplest form of propagation makes each arc consistent (i.e., X $\rightarrow$ Y is consistent iff for every value x of X there is some allowed y)
  - If X loses a value, its neighbors need to be rechecked;
  - Can be run as a preprocessor or after each assigment

Early Detection Example



Arc Consistency Algorithm

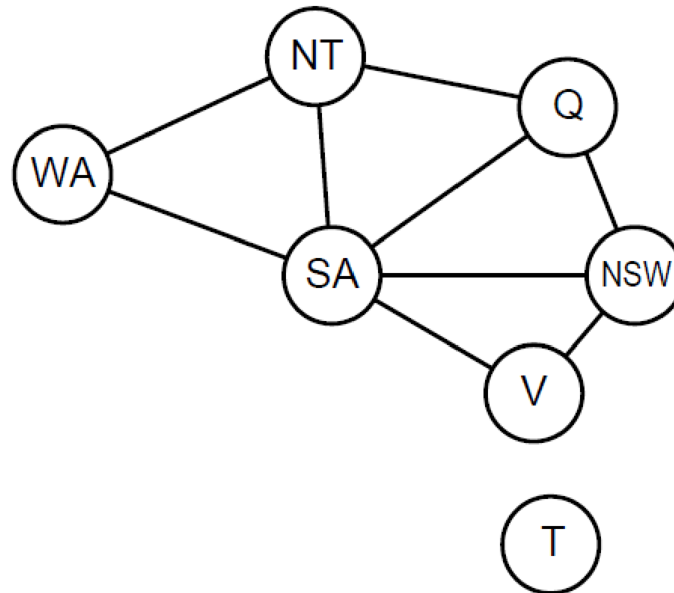
```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (Xi, Xj) ← REMOVE-FIRST(queue)
      if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do
            add (Xk, Xi) to queue
```

```
function REMOVE-INCONSISTENT-VALUES( Xi, Xj) returns true iff succeeds
   removed ← false
   for each x in DOMAIN[Xi] do
      if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
         then delete x from DOMAIN[Xi];  removed ← true
   return removed
```

Problem Structure

- Tasmania and mainland are independent subproblems;
- They are identifiable as connected components of constraint graph



Thank you!