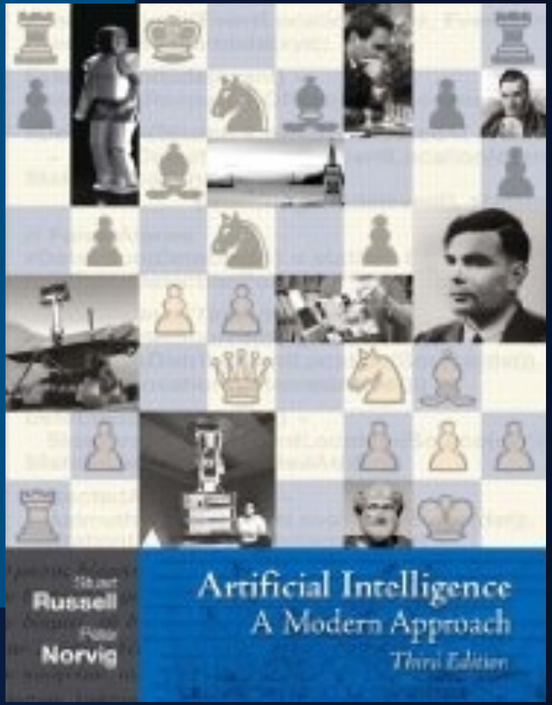
EDWARD TSANG
Foundations of Constraint Satisfaction
Edited by Thom Fruehwirth



AI Fundamentals: Constraints Satisfaction Problems

Maria Simi

Constraints satisfaction

LESSON 3

– SEARCHING FOR SOLUTIONS

Searching for solutions

Most problems cannot be solved by constraint propagation alone. In this case we must search for solutions or combine constraint propagation with search.

A classical formulation of CSP as search problem is the following:

1. States are partial assignments
2. Initial state: the empty assignment
3. Goal state: a complete assignment satisfying all constraints
4. Action: assign to a unassigned variable x_i a value in D_i

Branching factor = $|D_1| \times |D_2| \times \ldots \times |D_n|$?

Assume d the maximum cardinality

Top level $n \times d$, then $(n-1) \times d$, … $1 \times d$ $\rightarrow$ $n! \times d^n$ leaves

We can do better …

CSP as search problems

We can exploit **commutativity**. A problem is commutative if the order of application of any given set of actions has no effect on the outcome. In this case the order of variable assignment does not change the result.

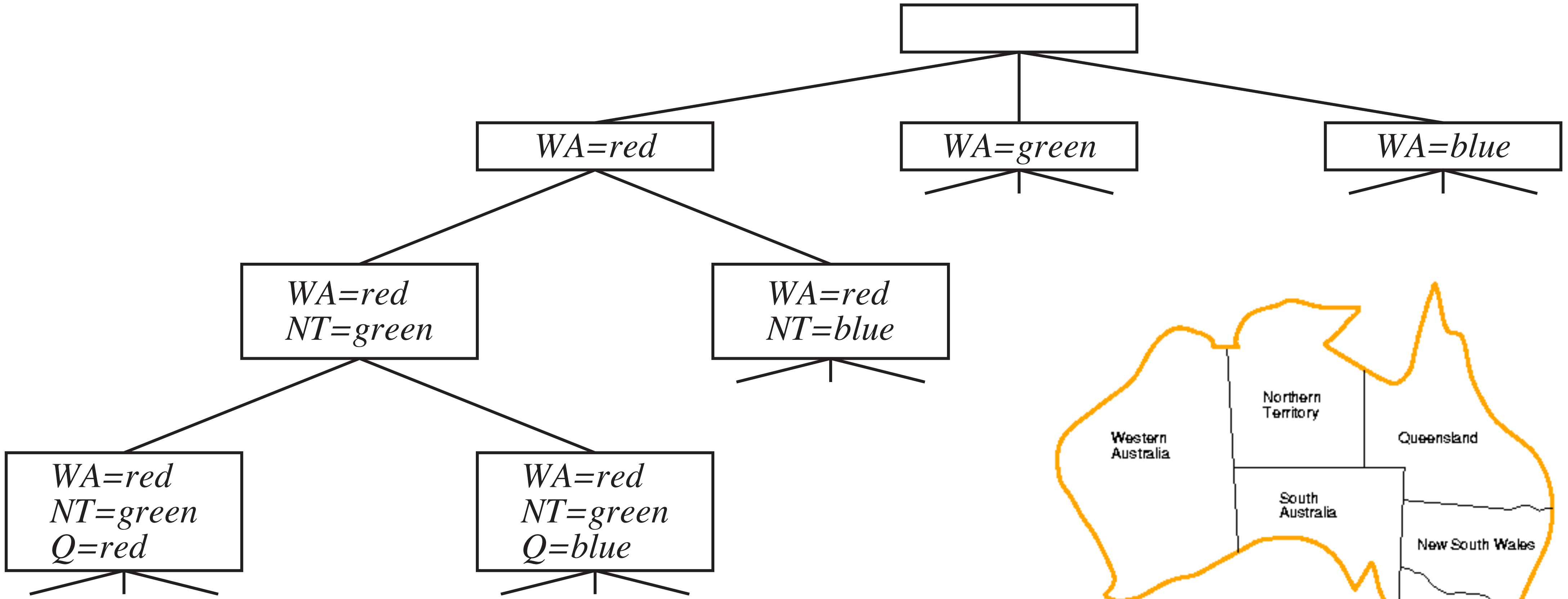
1. We can consider a **single variable** for assignment at each step, so the branching factor is d and the number of leaves is d^n
2. We can also exploit **depth limited search**: *backtracking search* with depth limit n, the number of variables.

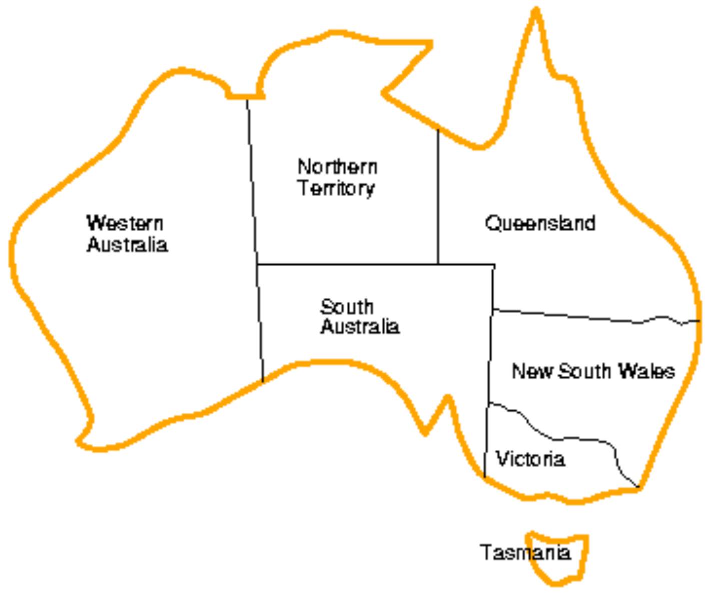
Search strategies:

Generate and Test. We generate a full solution and then we test it. Not the best.

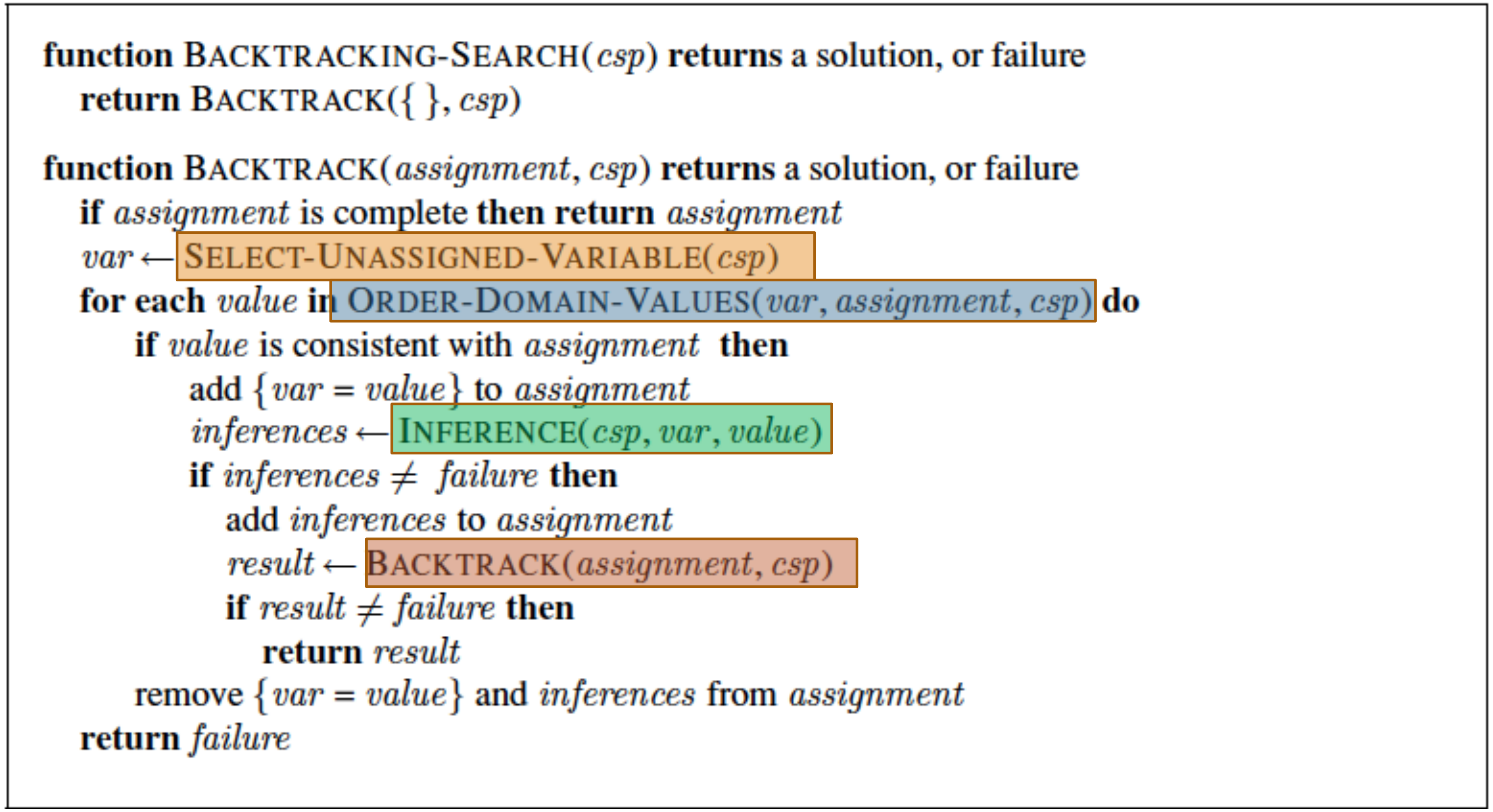
Anticipated control. After each assignment we check the constraints; if some constraints is violated, we backtrack to previous choices (undoing the assignment).

Backtracking search for map coloring





Backtracking search algorithm



```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
     if value is consistent with assignment then
        add {var = value} to assignment
        inferences ← INFERENCE(csp, var, value)
        if inferences ≠ failure then
          add inferences to assignment
          result ← BACKTRACK(assignment, csp)
          if result ≠ failure then
             return result
     remove {var = value} and inferences from assignment
  return failure
```

Heuristics and search strategies for CSP

1. SELECT-UNASSIGNED-VARIABLE: Which variable should be assigned next?
2. ORDER-DOMAIN-VALUES: in which order should the values be tried?
3. INFERENCE: what inferences should be performed at each step in the search? Techniques for **constraint propagation** (local consistency enforcement) can be used.
4. BACKTRACKING: where to back up to? When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure? Forms of **intelligent backtracking.**

Choosing the next variable to assign

Dynamic ordering is better.

1. **Minimum-Remaining-Values (MRV) heuristic:** Choosing the variable with the fewest “legal” remaining values in its domain.

   Also called **most costrained variable** or **fail-first** heuristics, because it helps in discovering inconsistencies earlier.

2. **Degree heuristic**: select the variable that is involved in the largest number of constraints on other unassigned variables. To be used when the size of the domains is the same. The degree heuristic can be useful as a *tie-breaker* in connection with MRV.

   Example: which variable to choose at the beginning in the map coloring problem?

Choosing a value to assign

Once a variable has been selected, the algorithm must decide on the order in which to assign values to it.

1. **Least-constraining-value**: prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph. The heuristic is trying to leave the maximum flexibility for subsequent variable assignments.

Note:

In the choice of variable, a *fail-first* strategy helps in reducing the amount of search by pruning larger parts of the tree earlier. In the choice of value, a *fail-last* strategy works best in CSPs where the goal is to find any solution; not effective if we are looking for all solutions or no solution exists.

Interleaving search and inference

- One of the simplest forms of inference/constraint propagation is **Forward Checking (FC)**
- Whenever a variable X is assigned, the forward-checking process establishes arc consistency of X for the arcs connecting neighbor nodes:

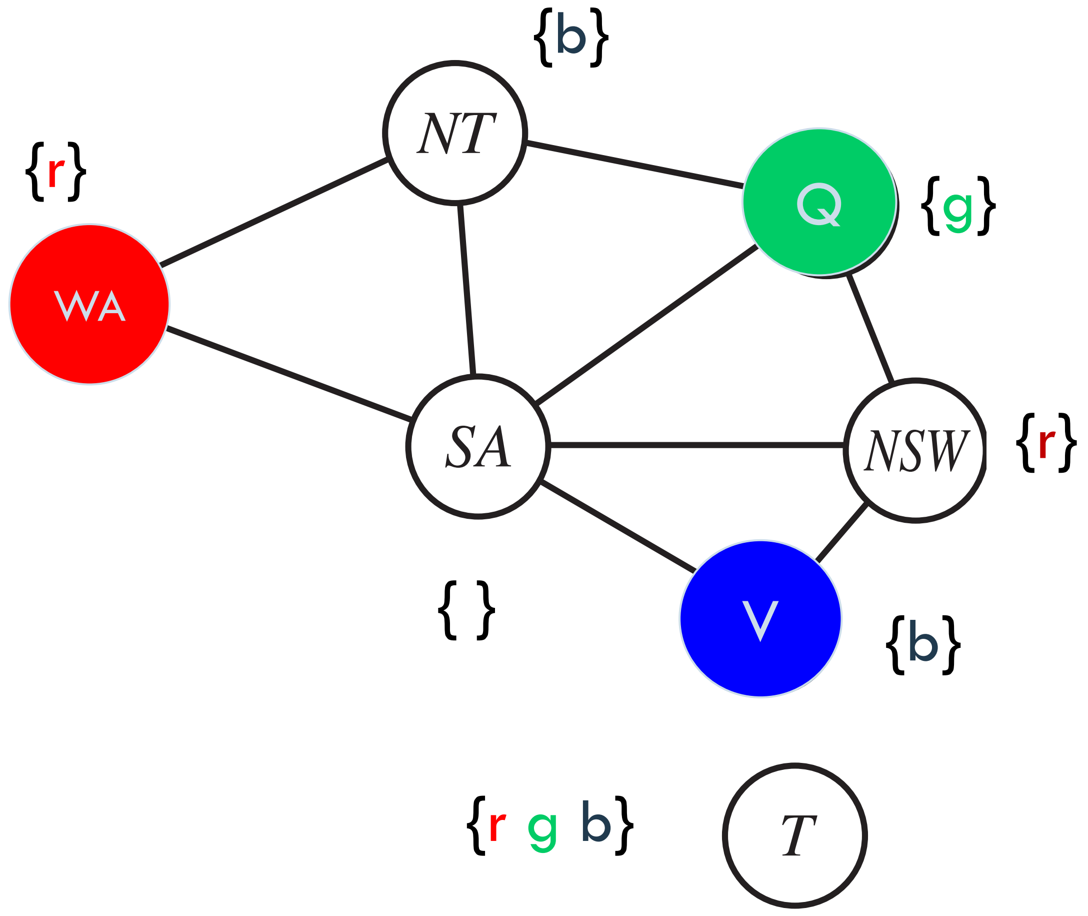
  for each unassigned variable Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value assigned to X.
- *Forward checking* is a form of efficient constraint propagation and is weaker than other forms of inference.

FC applied to map coloring

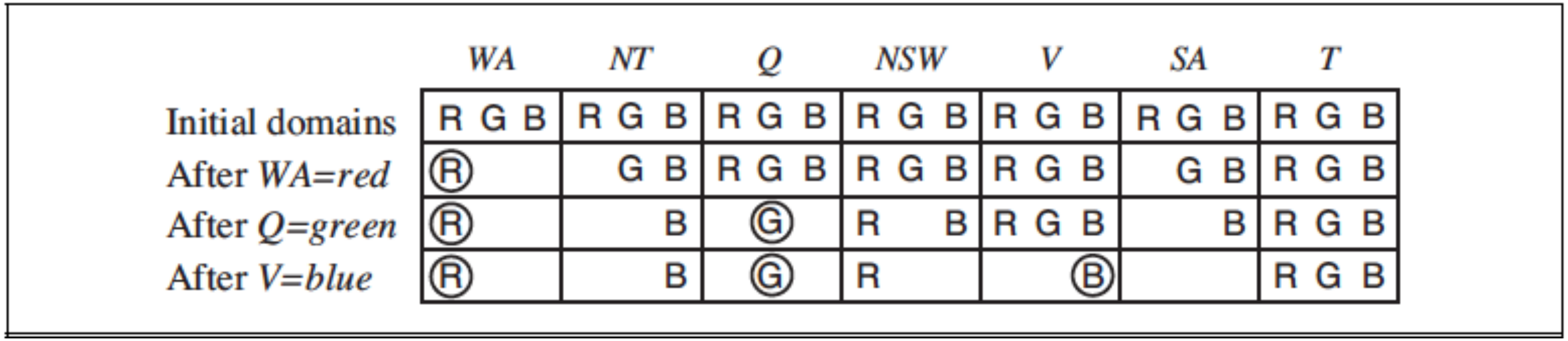
$WA = \mathrm{r}$

$Q = \mathrm{g}$

$V = \mathrm{b}$



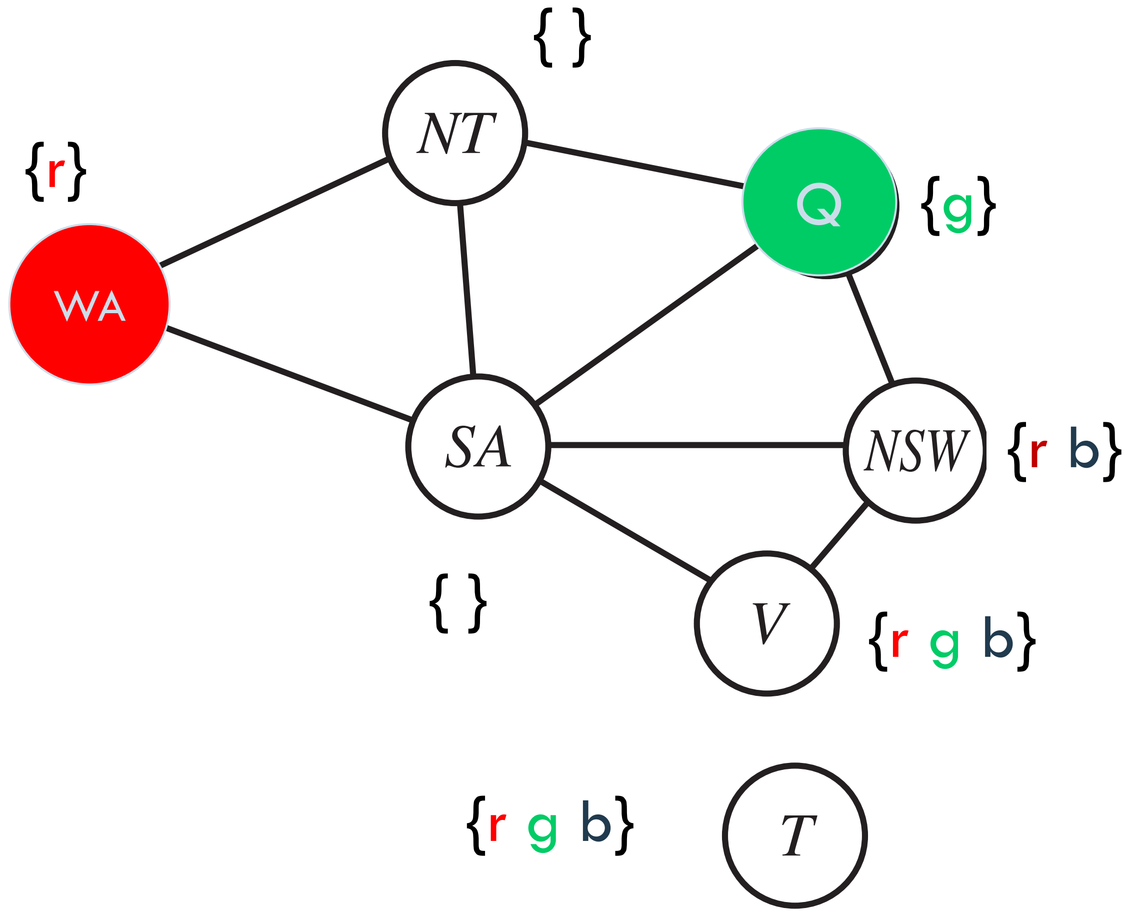
The same example showing the progress



	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	Ⓡ	B	Ⓖ	R B	R G B	B	R G B
After *V=blue*	Ⓡ	B	Ⓖ	R	Ⓑ		R G B

Maintaining Arc Consistency (MAC)

The inference procedure calls AC-3, but instead of a queue of all arcs in the CSP, we start with only the arcs (X_j, X_i) for all X_j that are unassigned variables that are neighbors of X_i.

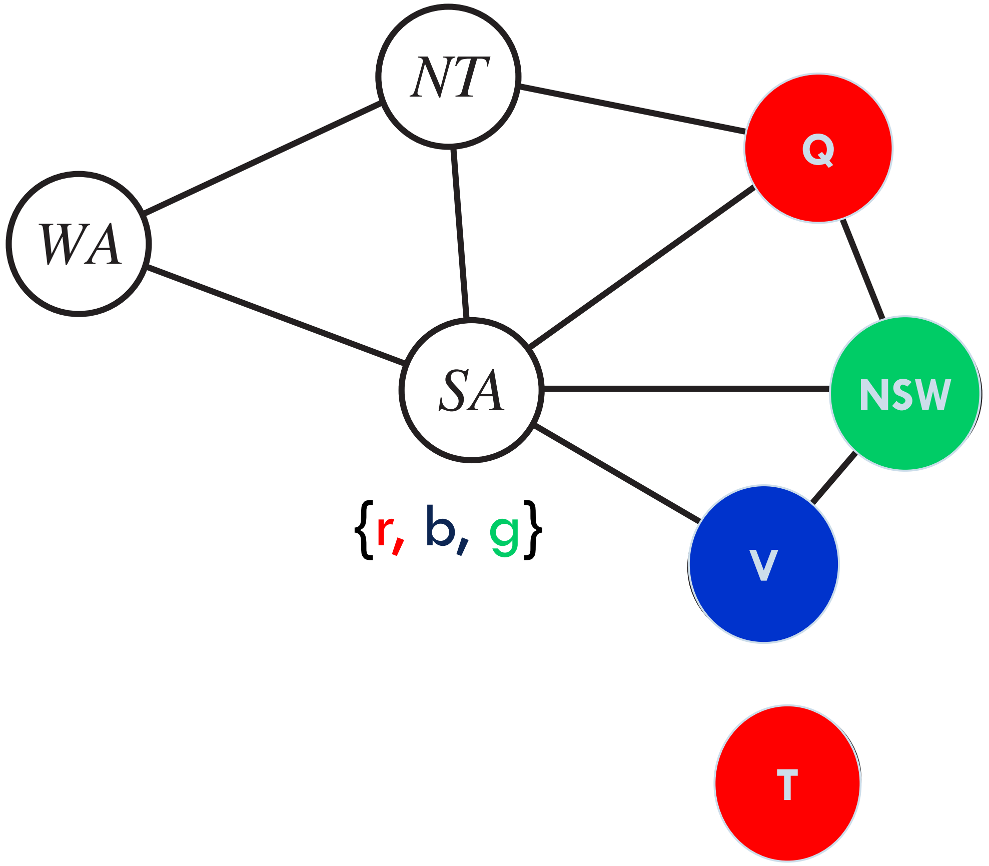


$WA = \mathrm{r}$

$Q = \mathrm{g}$

Chronological backtracking

- With normal/**chronological** backtracking with ordering *Q, NSW, V , T, SA, WA, NT*.
- Suppose we have assigned: {*Q*=red, *NSW*=green, *V*=blue, *T*=red} and we consider a value for SA
- We fail repeatedly, in fact no value is good in this situation
- Chronological backtracking tries all the values for Tasmania, the last variable, but we continue to fail. *Trashing behavior.*
- Tasmania has nothing to do with SA, nor the other variables.

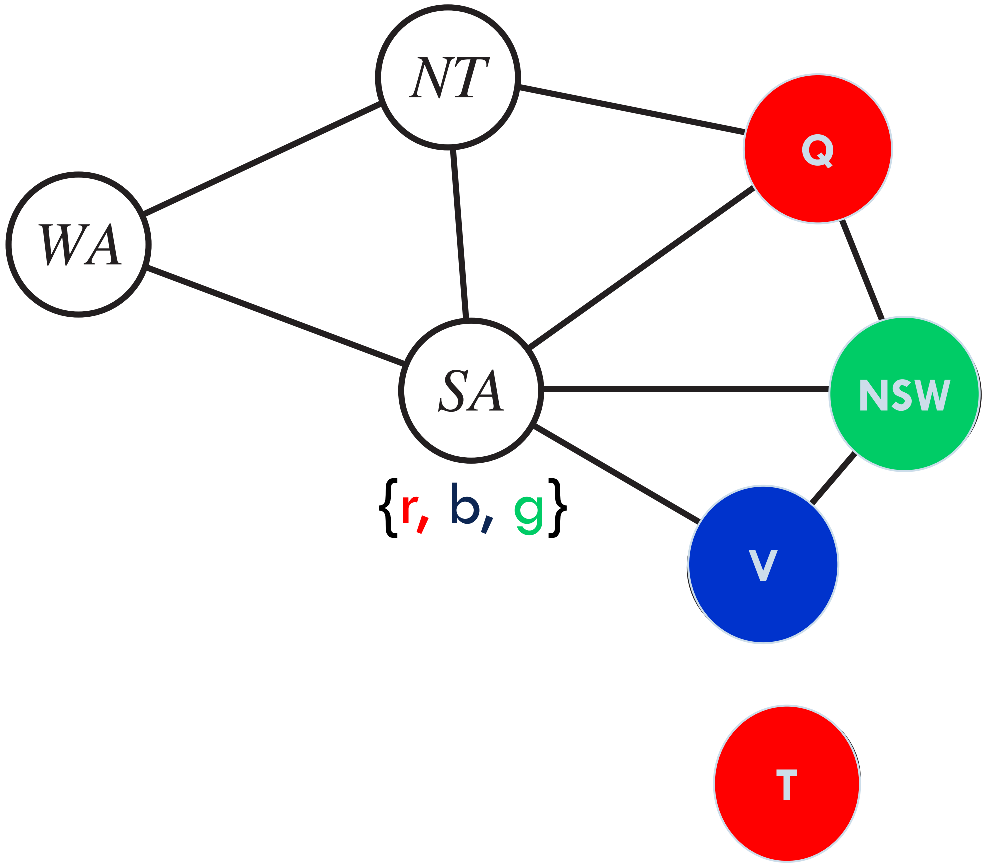


Intelligent backtracking: looking backward

The trick is to consider alternative values only for the assigned variables which are responsible for the violation of constraints: the **conflict set**. In this case $\{Q, NSW, V\}$,

The **backjumping method** backtracks to the most recent assignment in the conflict set; in this case, backjumping would jump over Tasmania and try a new value for V.

This strategy is not useful if we do FC or MAC as inconsistency is detected earlier.



Conflict directed backjumping

Assume ordering *WA, NSW, T, NT, Q, V, SA*.

$WA =$ red, $NSW =$ red and $T =$ red.

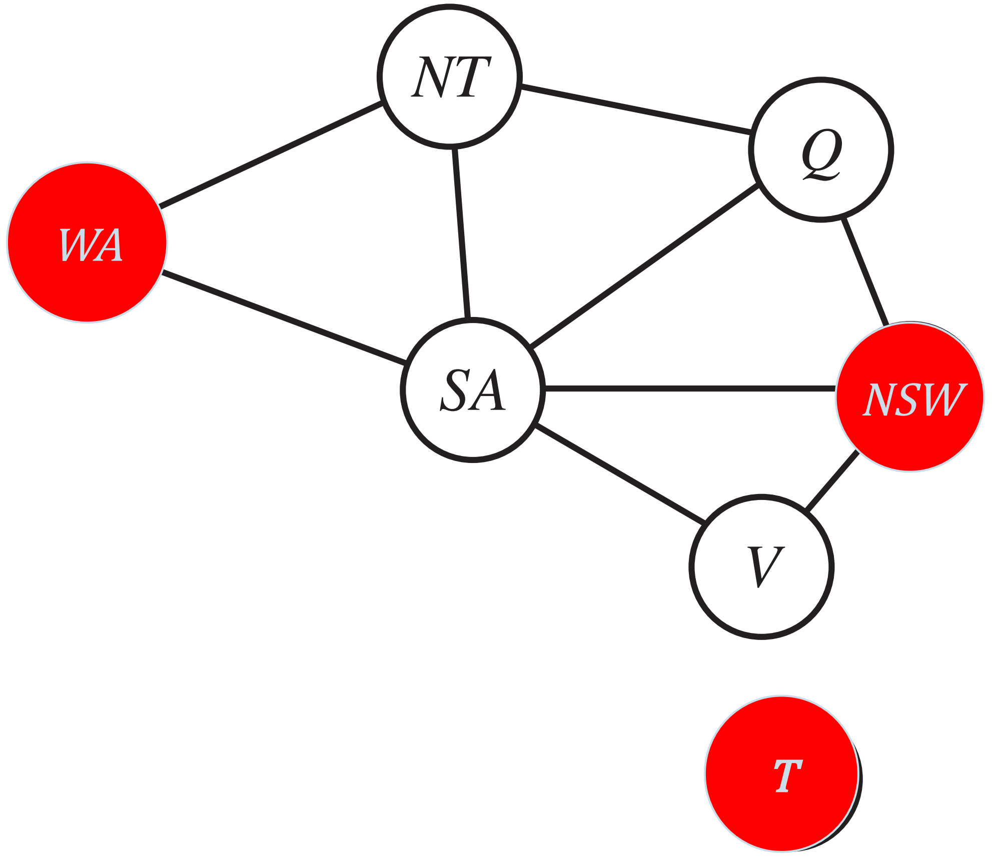
NT, Q, V, SA together do not have solutions. Where to backjump if we discover inconsistency at NT?

NSW is the last responsible. The conflict set of NT, {WA}, does not help.

Rule for computing a more informative conflict set:

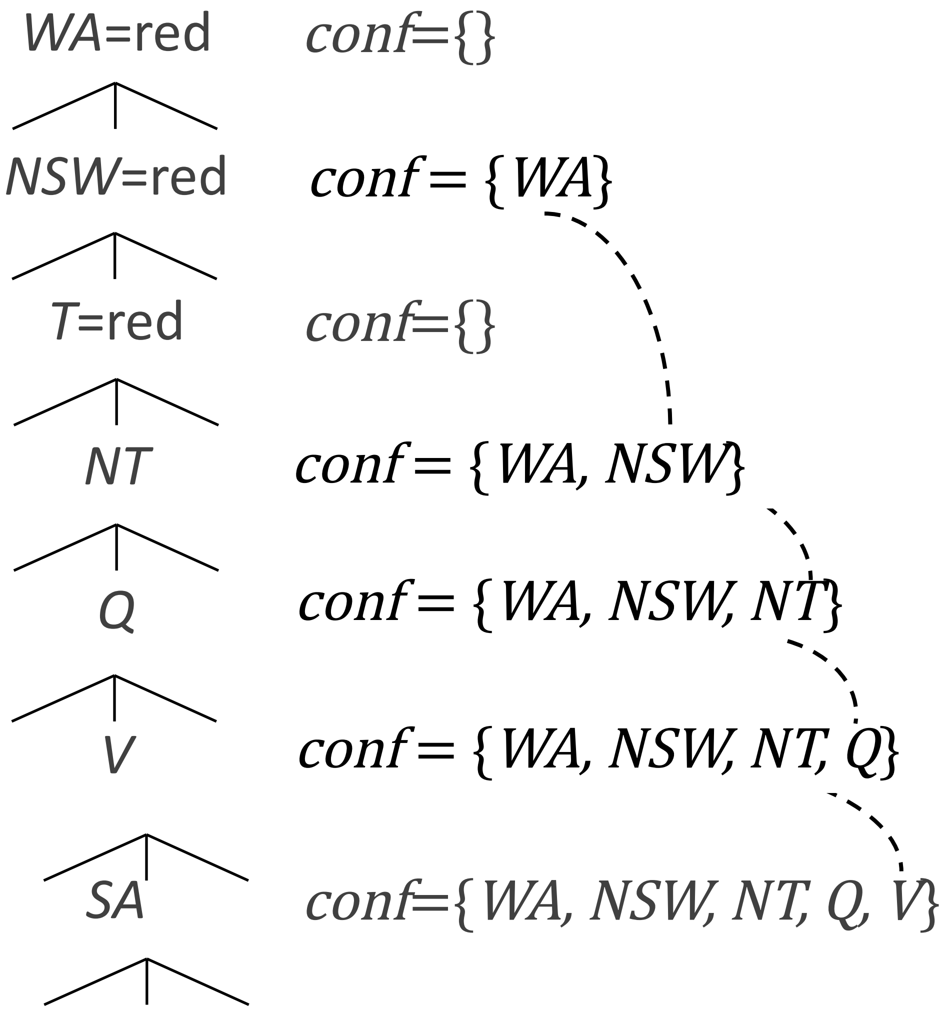
If every possible value for X_j fails, backjump to the most recent variable X_i in conf (X_j), and update its conflict set:

$$conf(X_i) \leftarrow conf(X_i) \cup conf(X_j) - \{X_i\}$$



Computing conflict sets

1. When SA fails, its conflict set is $\{WA, NSW, NT, Q, V\}$. We backjump to V. V's conflict set is extended with that of SA and becomes: $\{WA, NSW, NT, Q\}$.
2. When V fails, we backjump to Q. Q's conflict set is extended with that of V and becomes: $\{WA, NSW, NT\}$.
3. When Q fails, we backjump to NT and extend its conflict set $\{WA\}$ to $\{WA, NSW\}$.
4. When alsp NT fails we can backjump to NSW and update its conflict set to $\{WA\}$



Constraint learning

When the search arrives at a contradiction, we know that some subset of the conflict set is responsible for the problem.

Constraint learning is the idea of finding a minimum set of variables from the conflict set that causes the problem.

This set of variables, along with their corresponding values, is called a **no-good.**

We record the no-good, either by adding a new constraint to the CSP or by keeping a separate cache of no-goods. This way when we encounter a nogood state again we do not need to repeat the computation.

The state {WA = red , NT = green, Q = blue} is a *no-good*. We avoid repeating that assignment.

Local search methods

- They require a **complete state formulation** of the problem: all the elements of the solution in the current state. For CSP a complete assignment.
- They keep in memory only the current state and try to improve it, iteratively.
- They do not guarantee that a solution is found even if it exists (they are **not complete**). They cannot be used to prove that a solution doe not exist.

To be used when:

- The search **space is too large** for a systematic search and we need to be very efficient in time and space.
- We need to provide a solution but it is not important to produce the set of actions leading to it (the *solution path*).
- We know in advance that **solutions exist**.

Local search methods for CSP

Local methods use a *complete-state* formulation: we start with a **complete random assignment** and we try to fix it until all the constraints are satisfied.

Local methods prove quite effective in large scale real problems where the solutions are densely distributed in the space such as, for example the ***n*-Queens problem**.

N-Queens as a CSP:

- Q_i: position of i-th queen in the i-th column of the board
- D_i: $\{1 \dots n\}$
- Costraints are "non-attack" constraints among pair of queens.
- *Initial state: a random configuration*
- *Action: change the value of one queen, i.e. move a queen within its column.*

A basic algorithm for local search [AIFCA]

```
function Local_search($V$, $Dom$, $C$) returns a complete & consistent assignment
    Inputs:
        $V$: a set of variables
        $Dom$: a function such that $Dom()$ is the domain of variable $X$
        $C$: set of constraints to be satisfied
    Local: $A$, an array of values indexed by variables in $V$, the assignment
    repeat until termination
        for each variable $X$ in $V$ do      # random initialization or random restart
            $A[X]$ := a random value in $Dom(X)$
        while not stop_walk() & $A$ is not a satisfying assignment do  # local search
            Select a variable $Y$ and a value $w \in Dom(Y)$      # consider successors
            Set $A[Y] := w$
        if $A$ is a satisfying assignment then
            return $A$
```

stop_walk() implements some stopping criterion

Specializations of the algorithm

- Extreme cases**:**
  - ✓ **Random sampling**: no walking is done to improve solution (*stop_walk* is always true), just generating random assignments and testing them
  - ✓ **Random walk**: no restarting is done (*stop_walk* is always false)
- Family of algorithms**:**
  - ✓ **Iterative best improvement**: choose the successor that most improves the current state according to an evaluation function *f* (*hill-climbing*, *greedy ascent/discent*). In In CSP, *f* = number of violated constraints or *conflicts*. We stop when *f*=0 or cycle.
  - ✓ Randomness can be used to **escape local minima**:
    - Random restart is a global random move
    - Random walk is local random move.
  - ✓ **Stochastic local search algorithms** combine Iterative best improvement with randomness

Variants of stochastic local search

Algorithms differ in how much work they require to guarantee the best improvement step. Which successor to select?

- **Most improving choice**

  selects a variable–value pair that makes the best improvement. If there are many such pairs, one is chosen at random. Needs to evaluate all of them.

- **Two stage choice**
  1. Select the variable that participates in most conflicts
  2. Select the value that minimizes conflicts or a random value

- **Any choice**
  1. Choose a conflicting variable at random/choose a conflict and a variable within it
  2. Select the value that minimizes conflicts or a random value

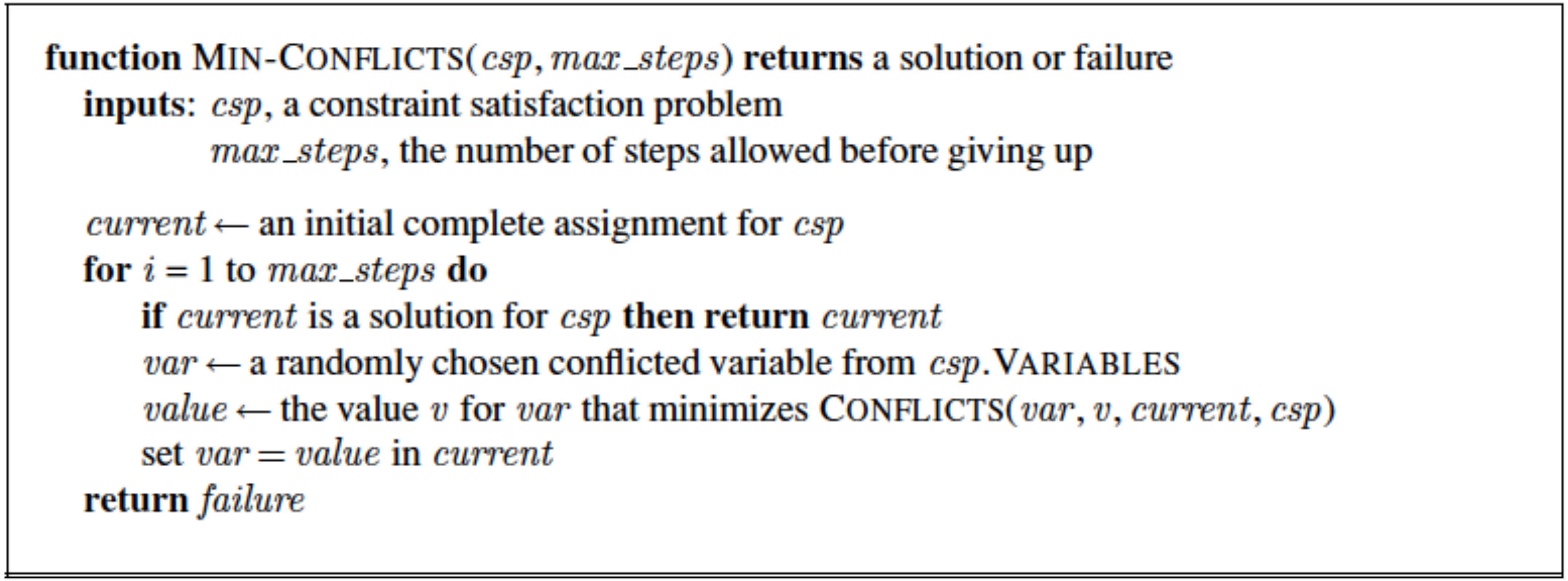
Min-conflict heuristics

All the local search techniques are candidates for application to CSPs, and some have proved especially effective.

Min-conflict heuristics is widely used and also quite simple:

- Select a variable at random among conflicting variables
- Select the value that results in the *minimum number of conflicts* with other variables.

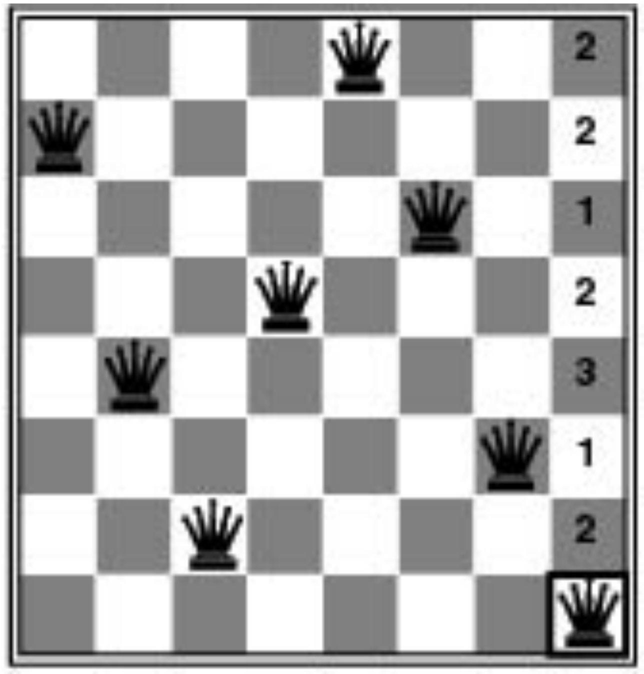
Min-conflicts algorithm

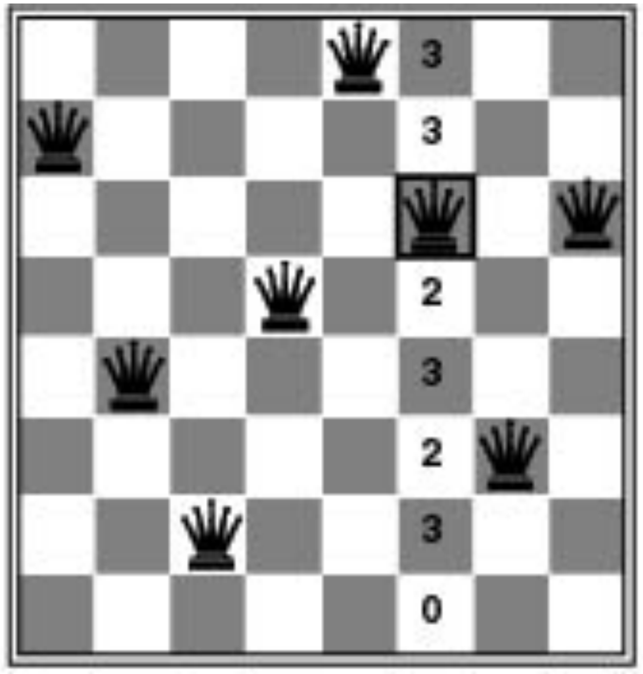


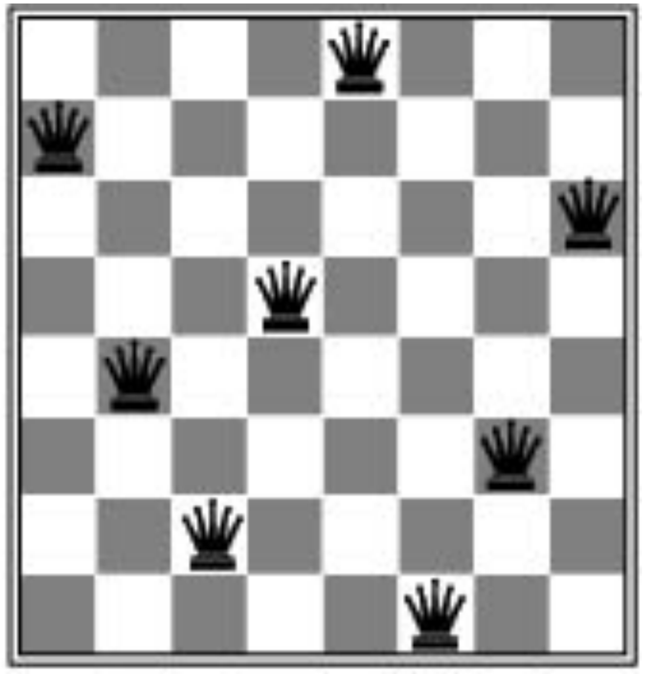
```
function MIN-CONFLICTS($csp$, $max\_steps$) returns a solution or failure
   inputs: $csp$, a constraint satisfaction problem
           $max\_steps$, the number of steps allowed before giving up

   $current \leftarrow$ an initial complete assignment for $csp$
   for $i$ = 1 to $max\_steps$ do
      if $current$ is a solution for $csp$ then return $current$
      $var \leftarrow$ a randomly chosen conflicted variable from $csp$.VARIABLES
      $value \leftarrow$ the value $v$ for $var$ that minimizes CONFLICTS($var$, $v$, $current$, $csp$)
      set $var = value$ in $current$
   return $failure$
```

Min-conflicts in action







The run time of min-conflicts is roughly independent of problem size.

It solves even the million-queens problem in an average of 50 steps.

Local search: improvements

The landscape of a CSP under the min-conflicts heuristic usually has a series of plateaux. Possible improvements are:

- **Tabu search**: local search has no memory; the idea is keeping a small list of the last *t* steps and forbidding the algorithm to change the value of a variable whose value was changed recently; this is meant to prevent cycling among variable assignments.
- **Constraint weighting**:

  can help concentrate the search on the important constraints. We assign a numeric weight to each constraint, which is incremented each time the constraint is violated. The goal is to minimize the weights of the violated constraints.

Local search: alternatives (see AIMA)

- **Simulated annealing**

  a tecnique for allowing downhill moves at the beginning of the algorithm and slowly *freezing* this possibiity as the algorithm progresses

- **Population based methods** (inspired by biological evolution):
  - ✓ *Local beam search*: proceed with the *k*-best successors (*k* = the beam width) according to the evaluation function.
  - ✓ *Stochastic local beam search*: selects *k* of the individuals at random with a probability that depends on the evaluation function; the individuals with a better evaluation are more likely to be chosen.
  - ✓ *Genetic algorithms …*

Local search methods and *online search*

Another advantage of local search methods is that they can be used in an **online setting** when the problem changes dynamically.

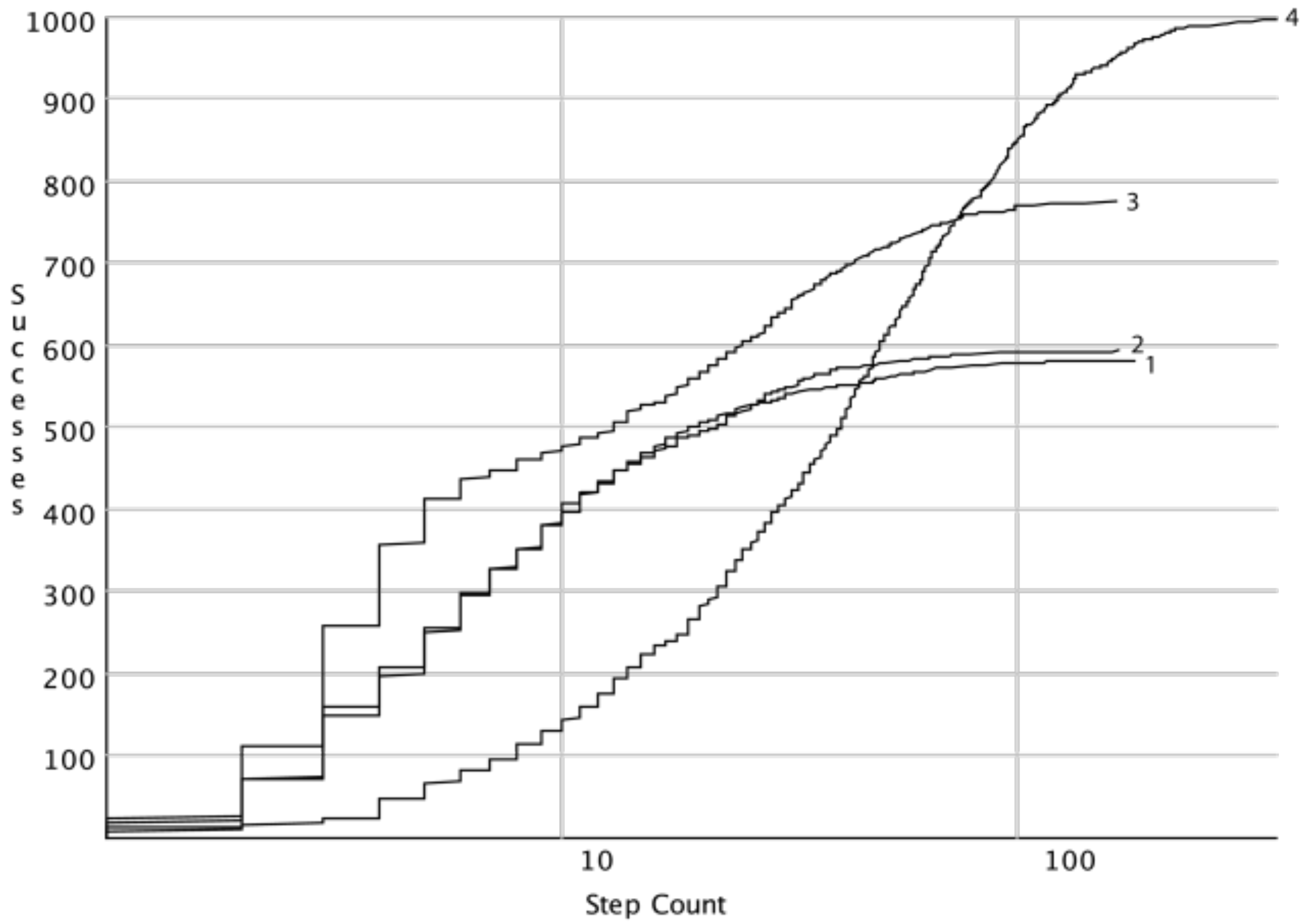
Consider complex flights schedules: a problem due to bad weather at one airport can render the schedule infeasible and require a rescheduling.

A local search methods are more effective as they try to repair the schedule with minimum variations, instead of producing from scratch a new schedule, which might be very different from the previous.

Evaluating Randomized Algorithms

- Randomized algorithms are difficult to evaluate since they give a different result and a different run time each time they are run. They must be run several times.
- Taking the **average run time** or **median run time** is ill defined. When the algorithm runs forever what do we do?
- One way to evaluate (or compare) algorithms for a particular problem instance is to visualize the **run-time distribution**, which shows the number of runs solved by the algorithm as a function of the number of steps (or run time) employed.

Run time distribution [AIFCA]



Increasing the probabilities of success

- A randomized algorithm that succeeds some of the time can be extended to an algorithm that succeeds more often by **running it multiple times**, using a **random restart**.
- An algorithm that succeeds with probability p, that is run n times or until a solution is found, will find a solution with probability

  $$1-(1-p)^n$$

  Note: $(1-p)^n$ is the probability of failing n times. Each attempt is independent.
- **Examples**:

  An algorithm with $p=0.5$ of success, tried 5 times, will find a solution around 96.9% of the time; tried 10 times it will find a solution 99.9% of the time.

  An algorithm with $p=0.1$, running it 10 times will succeed 65% of the time, and running it 44 times will give a 99% success rate.

Constraint Satisfaction

LESSON 3: THE STRUCTURE OF PROBLEMS

Independent sub-problems

The structure of the problem, as represented by the constraint graph, can be used to find solutions quickly. We will examine problems with a specific structure and strategies for improving the process of finding a solution.

The first obvious case is that of **independent subproblems**.

In the map coloring example, Tasmania is not connected to the mainland; coloring Tasmania and coloring the mainland are independent sub-problems—any solution the mainland combined with any solution for Tasmania yields a solution for the whole map.

Each **connected components** of the constraint graph corresponds to a sub-problem CSP_i.

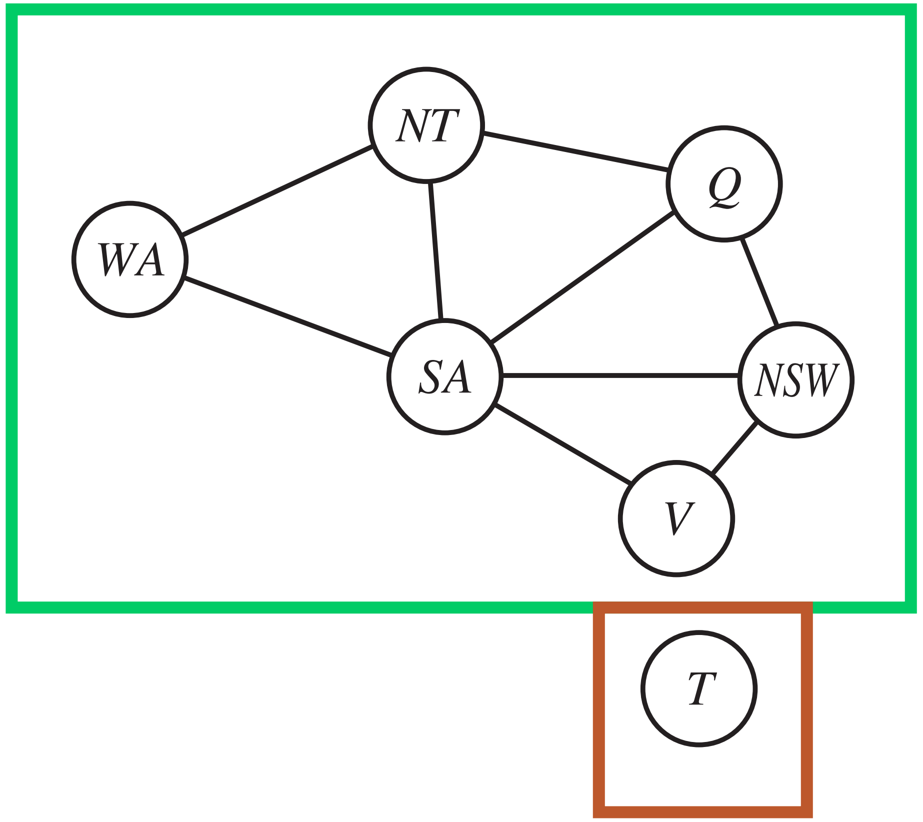
If assignment S_i is a solution of CSP_i, then $\bigcup_i S_i$ is a solution of $\bigcup_i CSP_i$

Independent sub-problems: complexity

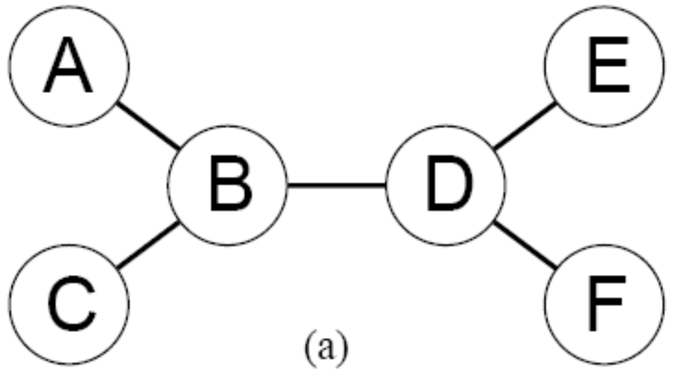
The saving in computational time is relevant

- *n* # variables
- *c* # variables for sub-problems
- *d* size of the domain
- *n/c* independent problems
- $O(d^c)$ complexity of solving one
- $O(d^c\, n/c)$ *linear* on the number of variables *n* rather than $O(d^n)$ *exponential*!

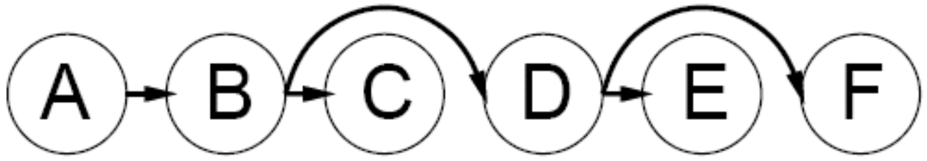
Dividing a Boolean CSP with 80 variables into four sub-problems reduces the worst-case solution time from the lifetime of the universe down to less than a second!!!



The structure of problems: trees



(a)



(b)

a) In a tree-structured constraint graph, two nodes are connected by only one path; we can choose any variable a the root of a tree. A in fig (b).

b) Chosen a variable as the root, the tree induces a **topological sort** on the variables. Children of a node are listed after their parent.

Directed Arc Consistency (DAC)

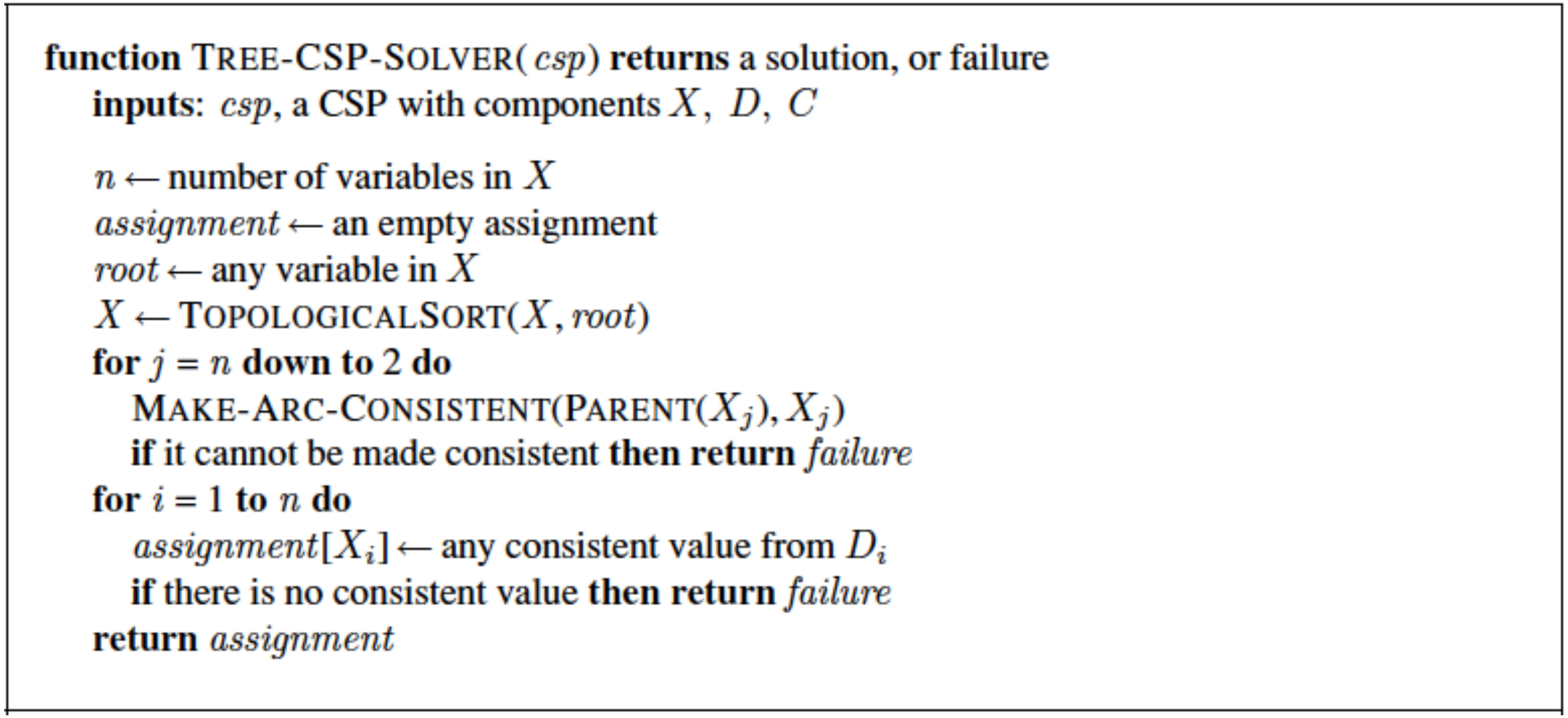
A CSP is defined to be **directed arc-consistent** under an ordering of variables X_1, X_2, … X_n if and only if every X_i is arc-consistent with each X_j for $j > i$.

We can make a tree-like graph *directed arc-consistent* in one pass over the *n variables*,; each step must compare up to d possible domain values for two variables, for a total time of O(nd^2).

Tree-CSP-solver:

1. Proceeding from X_n to X_2 make the arcs $X_i \rightarrow X_j$ *DAC* consistent by reducing the domain of X_i, if necessary. It can be done in one pass.
2. Proceeding from X_1 to X_n assign values to variables; *no need for backtracking* since each value for a father has at least one legal value for the child.

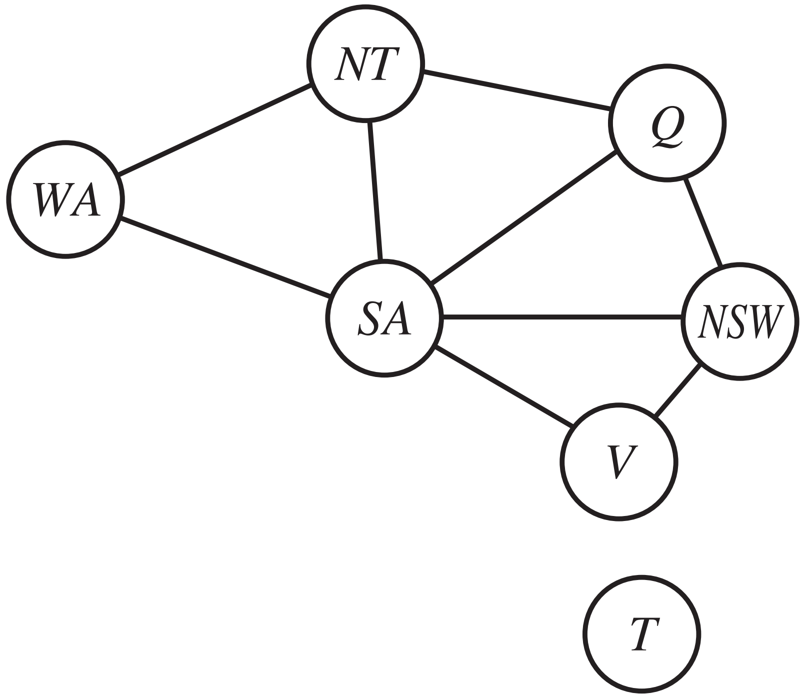
Tree-CSP-Solver algorithm

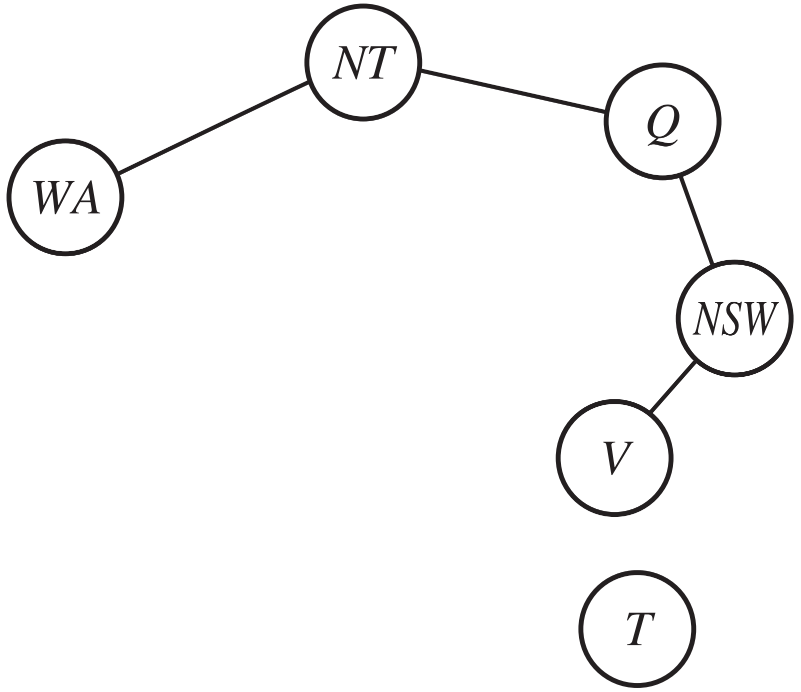


```
function TREE-CSP-SOLVER(csp) returns a solution, or failure
    inputs: csp, a CSP with components $X,\ D,\ C$

    $n \leftarrow$ number of variables in $X$
    $assignment \leftarrow$ an empty assignment
    $root \leftarrow$ any variable in $X$
    $X \leftarrow$ TOPOLOGICALSORT($X$, $root$)
    for $j = n$ down to 2 do
        MAKE-ARC-CONSISTENT(PARENT($X_j$), $X_j$)
        if it cannot be made consistent then return failure
    for $i = 1$ to $n$ do
        $assignment[X_i] \leftarrow$ any consistent value from $D_i$
        if there is no consistent value then return failure
    return assignment
```

Reducing graphs to trees





If we could delete South Australia, the graph would become a tree.

This can be done by fixing a value for SA (which one in this case does not matter) and removing inconsistent values from the other variables.

Cutset conditioning

In general we must apply a **domain splitting** strategy, trying with different assignments:

1. Choose a subset S of the CSP's variables such that the constraint graph becomes a tree after removal of S. S is called a **cycle cutset**.
2. For each possible consistent assignment to the variables in S :
   a. remove from the domains of the remaining variables any values that are inconsistent with the assignment for S
   b. If the remaining CSP has a solution, return it together with the assignment for S.

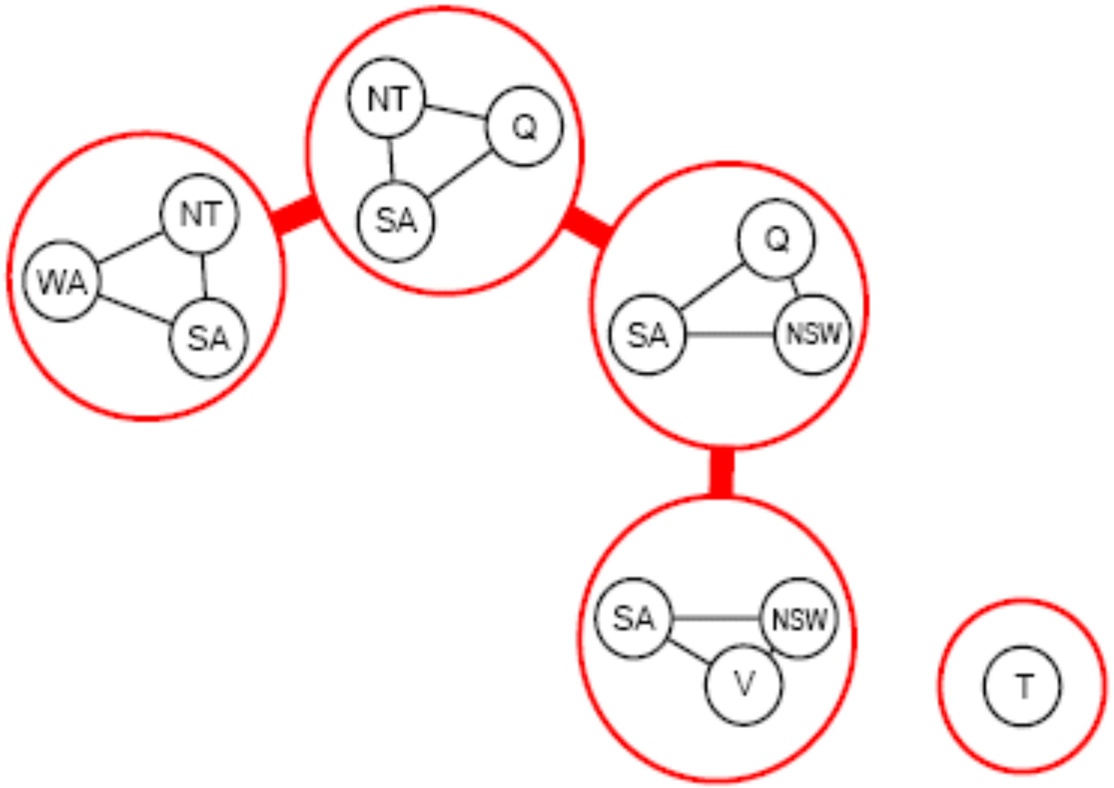
Time complexity: $O(d^c\,(n - c)d^2)$
where c is the size of the cycle cutset and d the size of the domain

We have to try each of the d^c combinations of values for the variables in S, and for each combination we must solve a tree problem of size $(n - c)$.

Tree decomposition

The approach consists in a **tree decomposition** of the constraint graph into a set of connected sub-problems.

Each sub-problem is solved independently, and the resulting solutions are then combined in a clever way



Properties of a tree decomposition

A tree decomposition must satisfy the following three requirements:

1. Every variable in the original problem appears in at least one of the sub-problems.
2. If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the sub-problems.
3. If a variable appears in two sub-problems in the tree, it must appear in every subproblem along the path connecting those sub-problems.

Conditions 1-2 ensure that all the variables and constraints are represented in the decomposition.

Condition 3 reflects the constraint that any given variable must have the same value in every sub-problem in which it appears; the links joining subproblems in the tree enforce this constraint.

Solving a decomposed problem

- We solve each sub-problem independently. If any problem has no solution, the original problem has no solution.
- Putting solutions together. We solve a meta-problem defined as follows:
  - Each sub-problem is "mega-variable" whose domain is the set of all solutions for the sub-problem

    Ex. $Dom(X_1)$ ={⟨WA=r, SA=b, NT=g⟩ ...} the 6 solutions to 1° subproblem
  - The constraints ensure that the subproblem solutions assign the same values to the the variables they share.

Ideally we should find, among many possible one, a tree decomposition with **minimal tree width** (the size of the largest subproblem – 1). This is NP-hard.

If w is the minimum tree width of the possible tree decompositions, the complexity is O(nd^{w+1}).

Simmetry

Simmetry is an important factor for reducing the complexity of CSP problems.

Value simmetry: the value does not really matters.

- *WA*, *NT*, and *SA* must all have different colors, but there are 3! = 6 ways to satisfy that. If *S* is a solution to the map coloring with *n* variables, there are *n*! solutions formed by permuting the color names.

Symmetry-breaking constraint:

- we might impose an arbitrary ordering constraint, $NT < SA < WA$, that requires the three values to be in alphabetical order; we get only one solution out of the 6.
- In practice, *breaking value symmetry* has proved to be important and effective on a wide range of problems.
- Symmetry is an important area of research in CSP.

Conclusions

- ✓ We have seen how to search efficiently for solutions:
- ✓ Algorithms for **local stochastic search** in large state spaces
- ✓ How we can exploit the structure of the problem
- ✓ CSP is a large field of study: many more things could be presented

Your turn

- ✓ Review ”population based methods” for CSP.
- ✓ Apply search techniques to the solution of some problem.
- ✓ Find a problem that can be solved with tree search and solve it with the tree search algorithm
- ✓ …

References

[AIMA] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach* (3rd edition). Pearson Education 2010. (Cap. 6)

[AIFCA] David L. Poole, Alan K. Mackworth. *Artificial Intelligence: foundations of computational agents* (2nd edition), Cambridge University Press, 2017– Computers. http://artint.info/2e/html/ArtInt2e.html (Cap 4)

[Tsang] Edward Tsang. *Foundations of Constraint Satisfaction*, Computation in Cognitive Science. Elsevier Science. Kindle Edition, 2014.