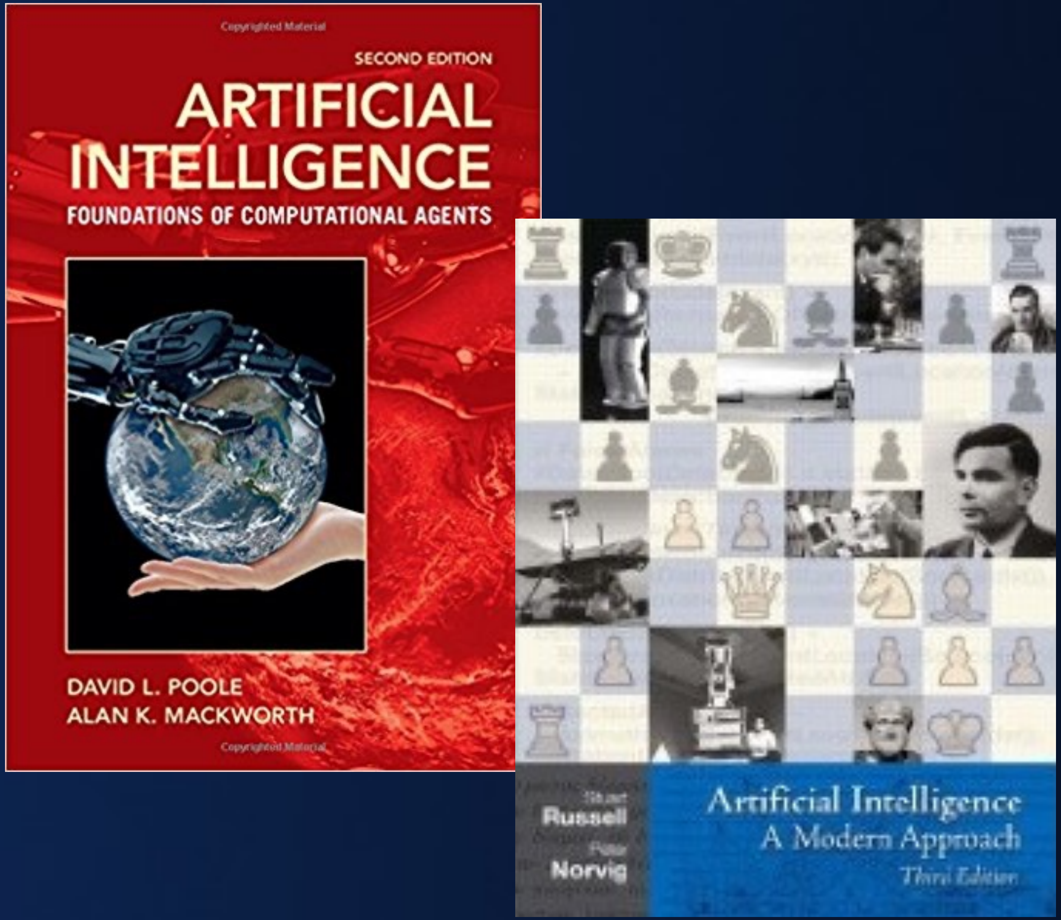


AI Fundamentals: Constraints Satisfaction Problems

Maria Simi

Summary

1. Searching for solutions
2. Local search methods for CSP (with review)
3. Exploiting the structure of problems

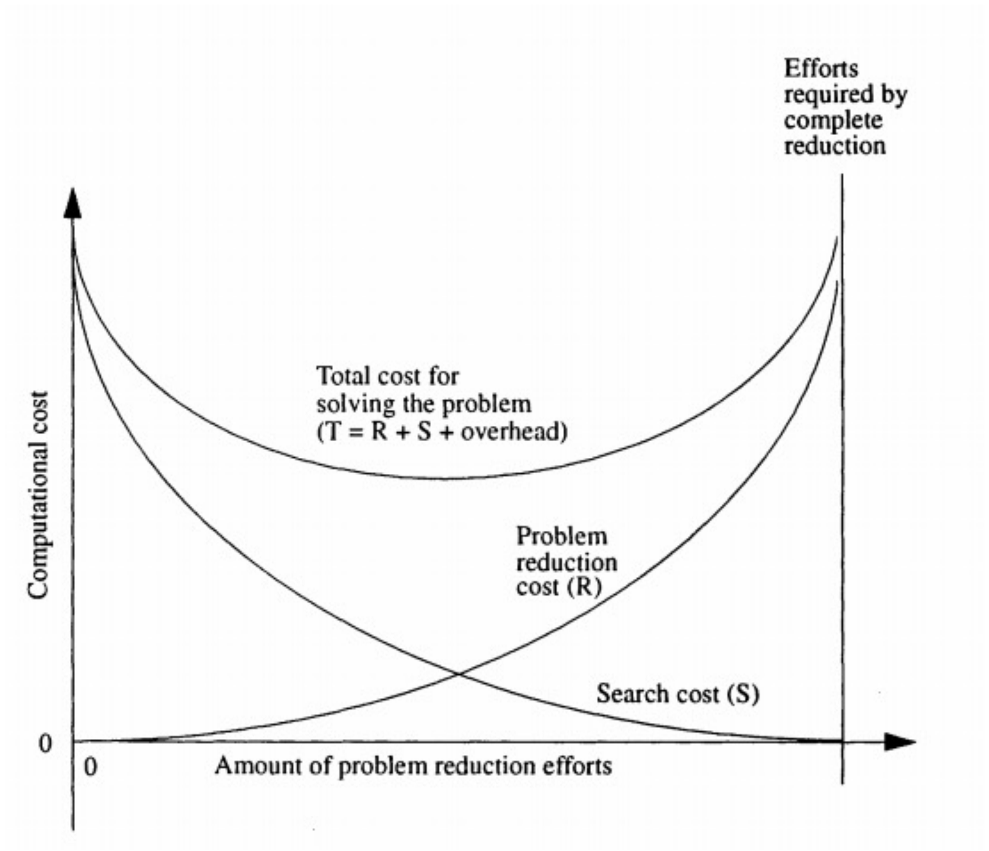
Reviews and exercises: next Thursday, October 7h

Assignments on this first part will close on October 5th.

Searching for solutions

LESSON 3 (AIMA CH 6, AI-FCA CH 4)

Combining problem reduction and search



Problem reduction techniques are used in combination with search.

The graph shows the cost of problem reduction (R) vs. cost of search (S).

The more effort one spends on problem reduction, the less effort one needs in searching [Tsang]

Searching for solutions

Most problems cannot be solved by problem reduction alone. In this case we must search for solutions and combine problem reduction with search. In this context we talk about **constraint propagation** or **inference**.

A classical *incremental* 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. Actions: assign to a specific unassigned variable x_i a value in D_i

Branching factor = d with d the maximum cardinality of the domains
Number of leaves: d^n and n the number of variables

Note 2*: n* is finite, the space graph is finite

CSP as search: simplifications

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.

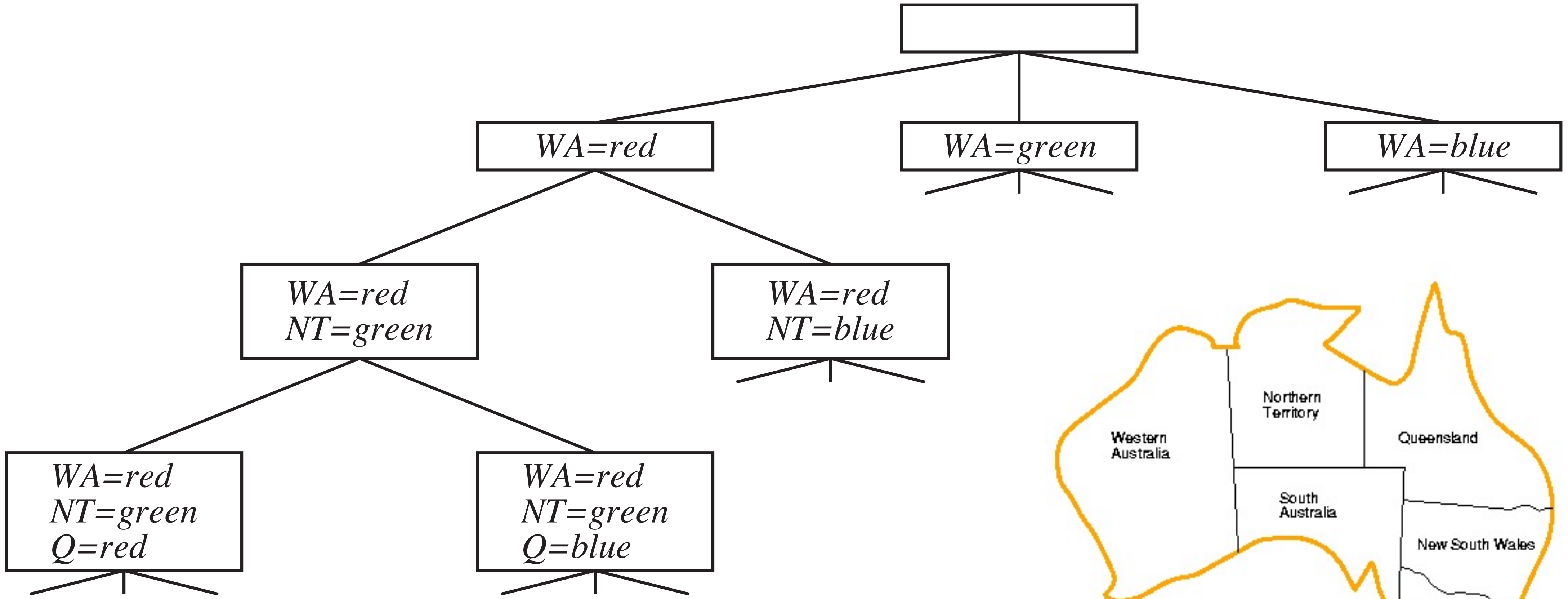
1. We can consider a **single variable** for assignment at each step, so the branching factor is d and the number of leaves is d^n
2. We can also exploit **depth limited search**: *backtracking search* with depth limit n, the number of variables.

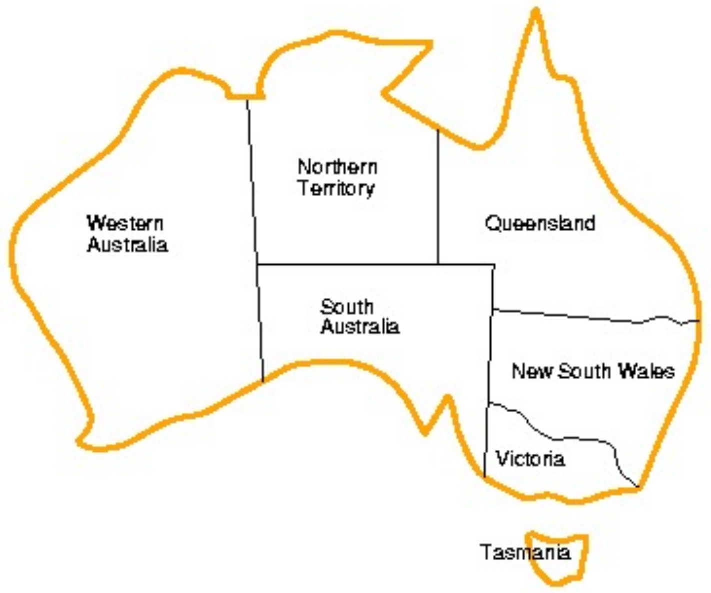
Search strategies:

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

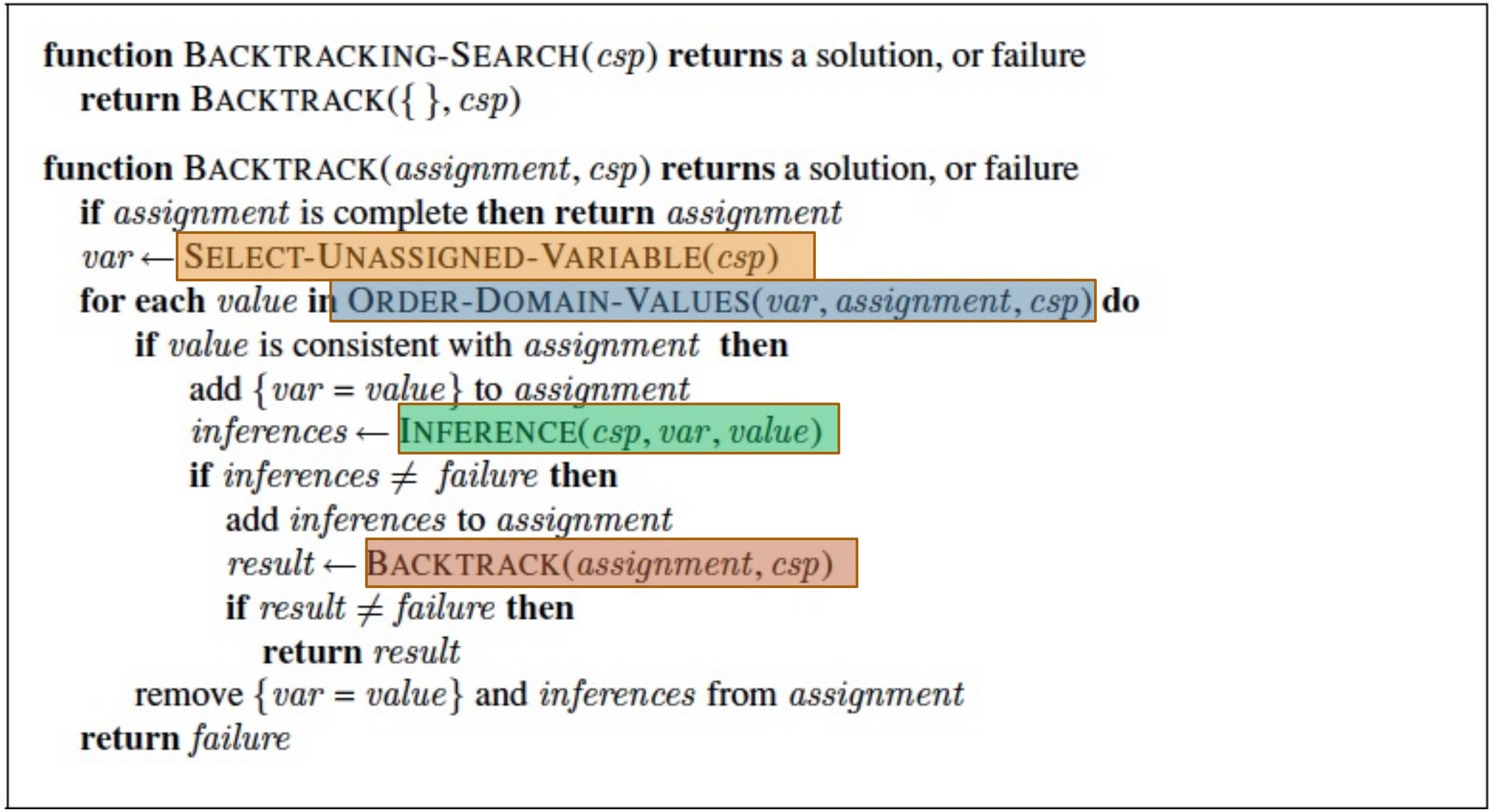
Anticipated control. After each assignment we check the constraints; if some constraints is violated, we backtrack to previous choices (undoing the assignment).

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:

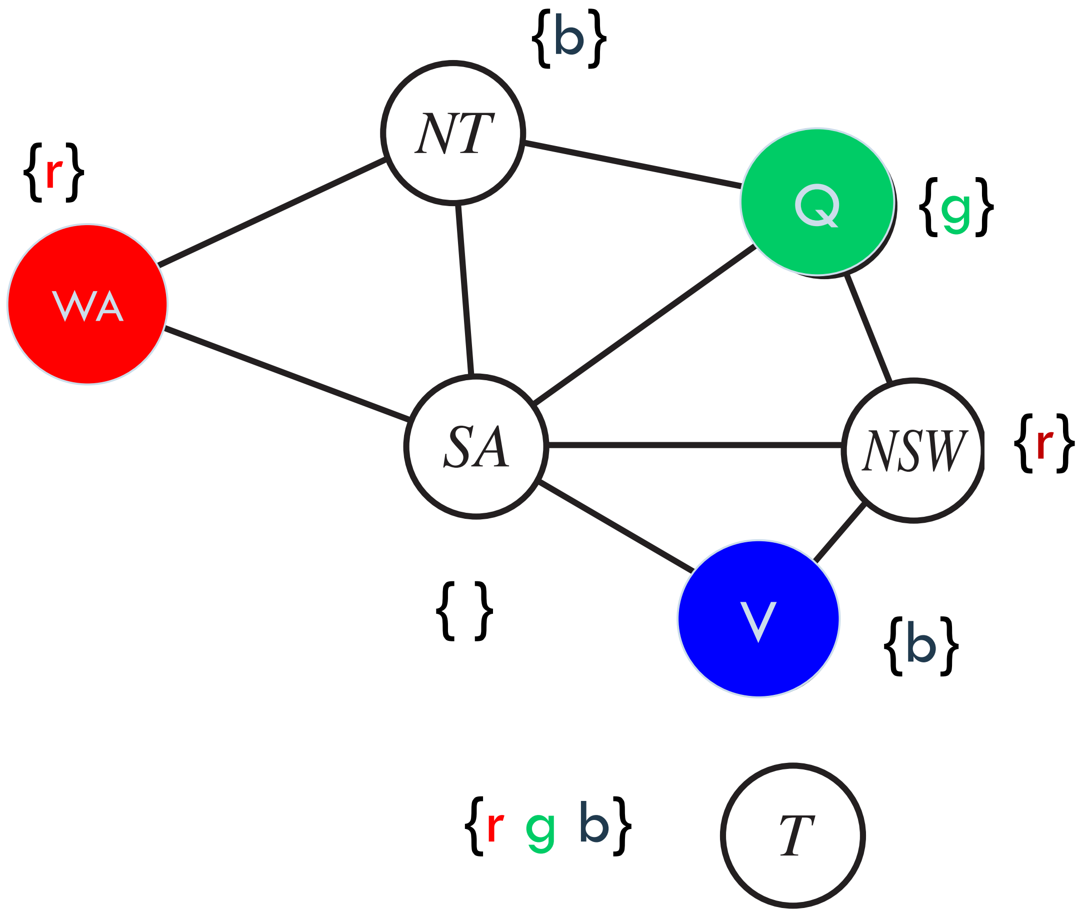
  for each unassigned variable Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value assigned to X.
- *Forward checking* is a form of efficient constraint propagation and is weaker than other forms of inference.

FC applied to map coloring

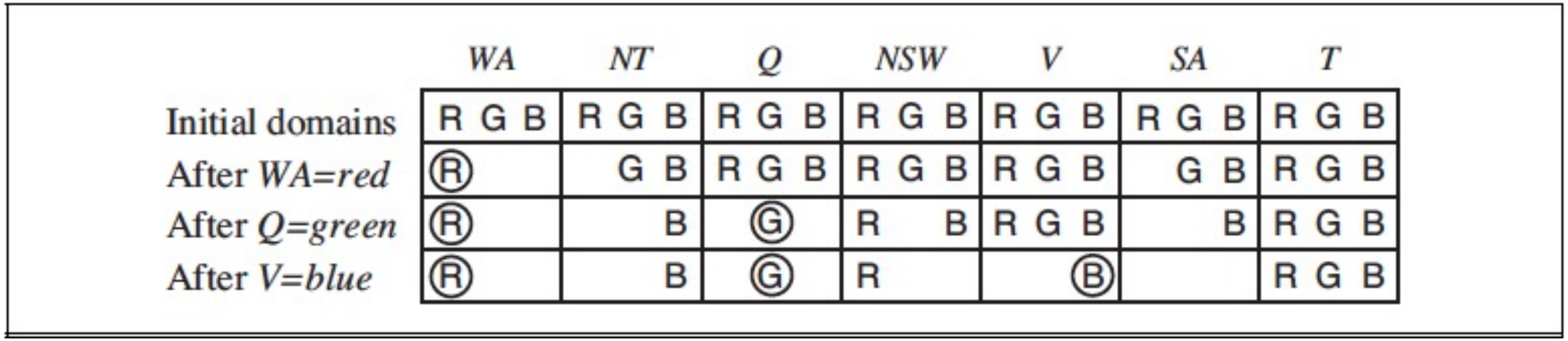
$WA = \mathrm{r}$

$Q = \mathrm{g}$

$V = \mathrm{b}$



The same example showing the progress



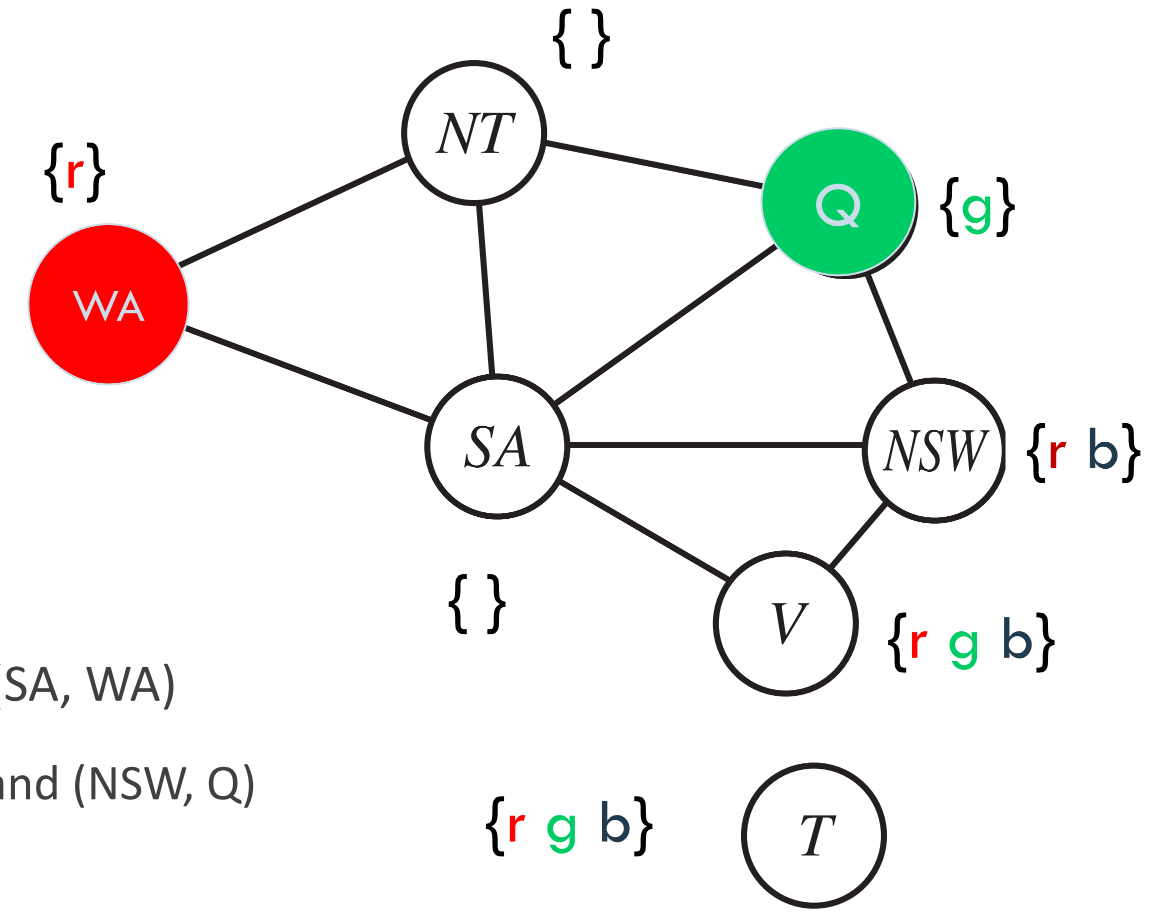
	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	Ⓡ	B	Ⓖ	R B	R G B	B	R G B
After *V=blue*	Ⓡ	B	Ⓖ	R	Ⓑ		R G B

Maintaining Arc Consistency (MAC)

We run AC-3 before starting search.

Suppose we assign a value to node X

The inference procedure calls AC-3, with initial arcs (Y, X) for all Y that are unassigned variables neighbors of X (the assigned variable)

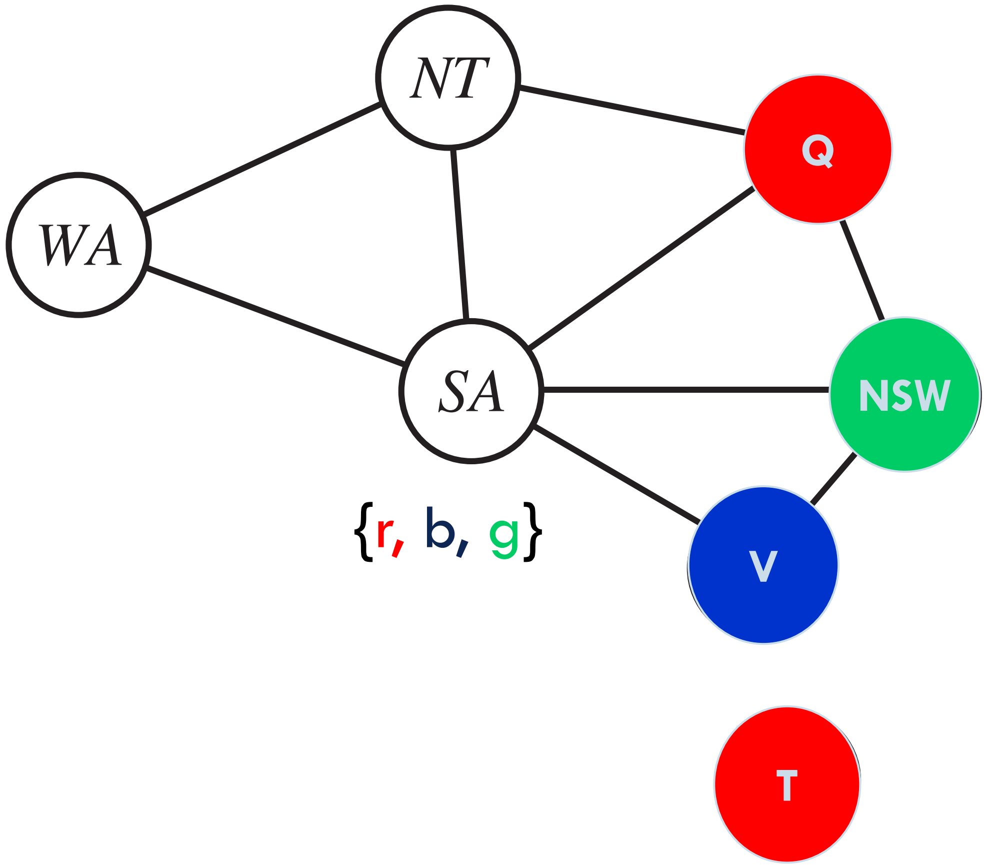


WA = r AC-3 initialized with (NT, WA) and (SA, WA)

$Q = \text{g}$ AC-3 initialized with (NT, Q), (SA, Q) and (NSW, Q)

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 to find an admissible value for SA.
- 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 failures.

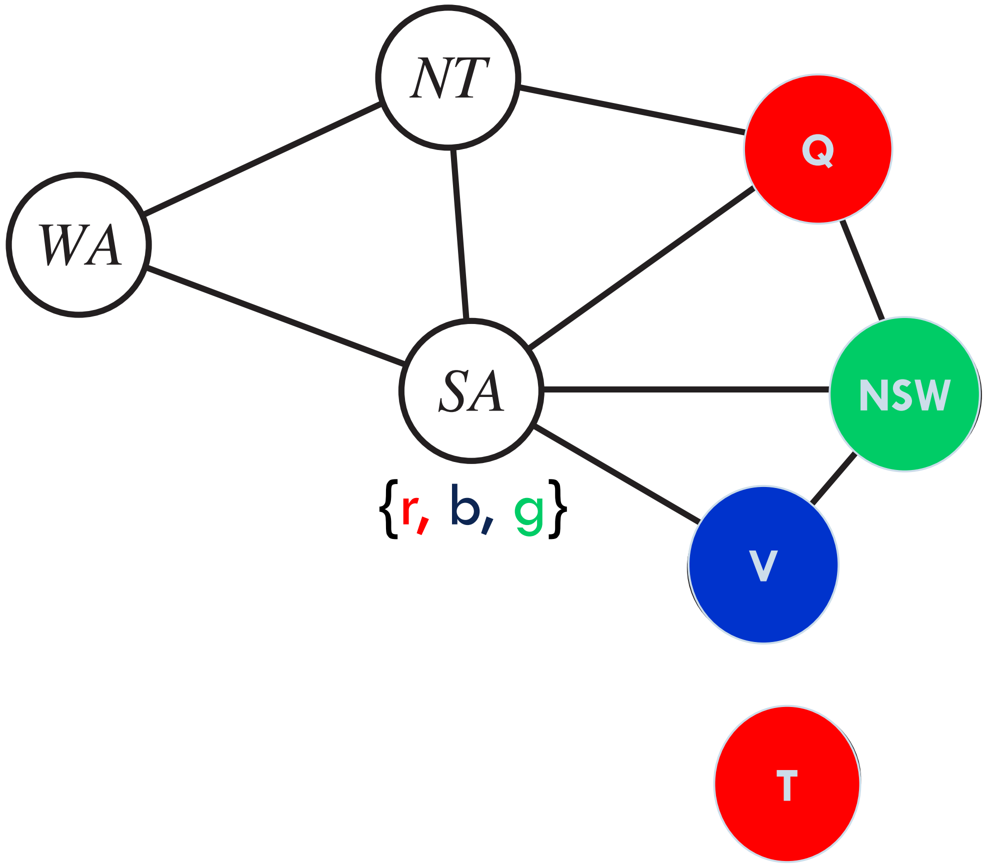


Intelligent backtracking: looking backward

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 back directly to V and try a new value for it.

Note: 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.

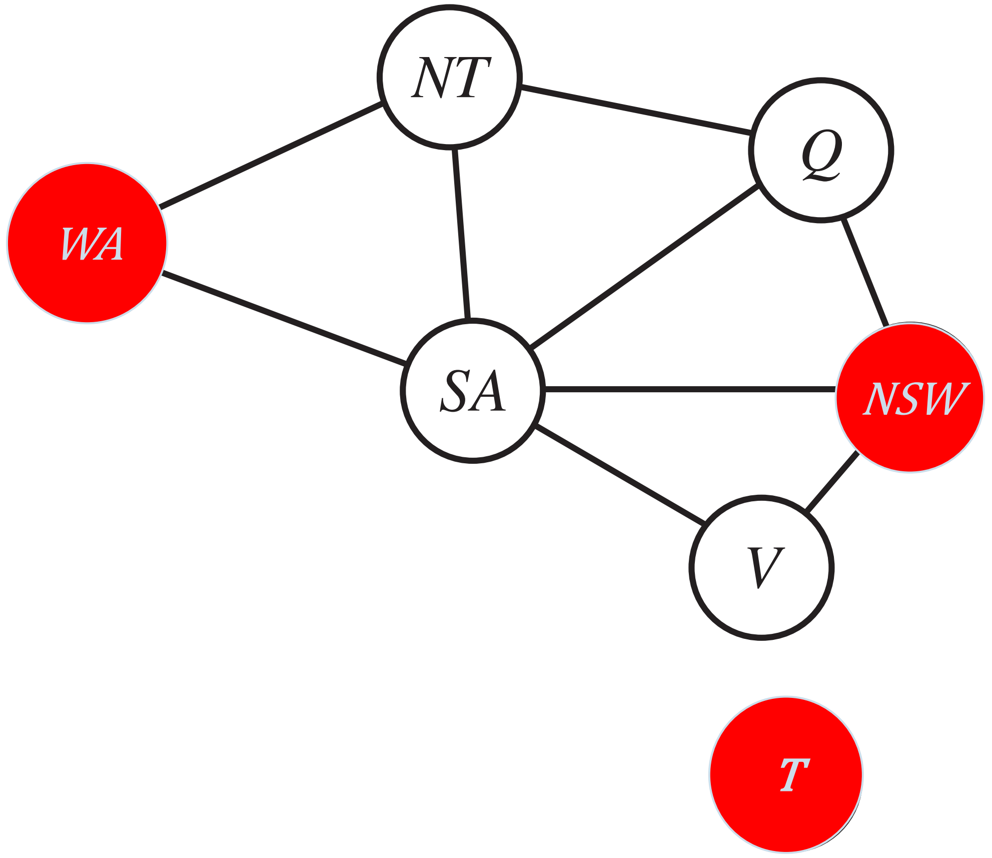
NT, Q, V, SA together do not have solutions. Where to backjump if we discover inconsistency at NT?

In this case *NSW* is the **last responsible**. The conflict set of NT, {WA}, does not include it, if we consider only adjacent variables.

The rule for computing the conflict set is:

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:***

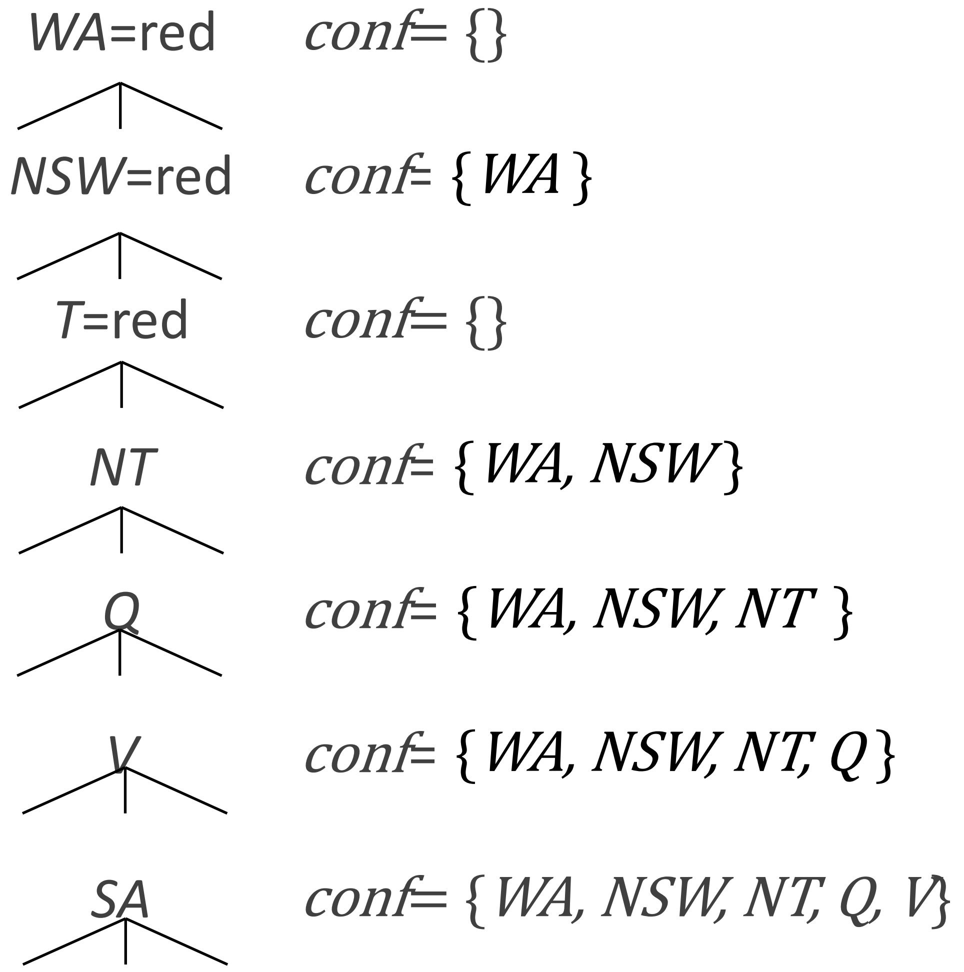
$$conf(X_i) \leftarrow conf(X_i) \cup conf(X_j) - \{X_i\}$$



Computing conflict sets

Ordering *WA, NSW, T, NT, Q, V, SA.*

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 also *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 *no-good* 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

REVIEW OF LOCAL SEARCH

Local search methods

- They require a **complete state formulation** of the problem: all the elements of the solution in the current state.
- 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 does not exist.

To be used when:

- The search **space is too large** for 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

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 … n}, row number
- 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.*
- *Evaluation of states*: number of conflicts, to be minimized,

A generic algorithm for local search in CSP [AI-FCA]

function Local_search(*V*, *Dom*, *C*) **returns** a complete & consistent assignment

Inputs: *V*: a set of variables

Dom: a function such that *Dom*(*x*) is the domain of variable *x*

C: set of constraints to be satisfied

Local: *A (complete assignement)* an array of values indexed by variables in *V*

```
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 $w \in Dom(y)$, $w \neq A[y]$        # a successors
        A[y] := w                                        # change a variable
    if A is a satisfying assignment then return A        # solution found
```

Specializations of the basic schema

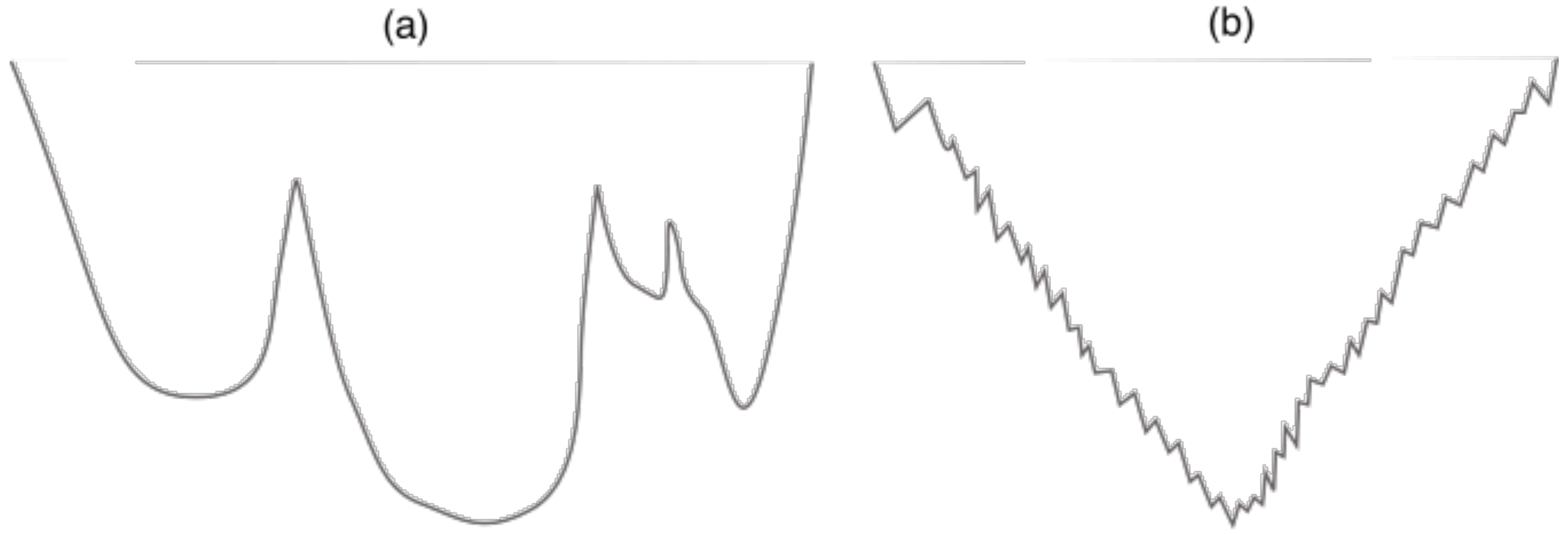
- Two extreme versions:
  - ✓ **Random sampling**: no walking is done to improve solution ($stop\_walk$ is always true), just generating random assignments and testing them (no walk)
  - ✓ **Random walk**: no restarting is done ($stop\_walk$ is always false)

Heuristic local search

- Heuristics algorithms
  - ✓ You can inject heuristics in the selection of the variable and the value by means of an **evaluation function.**

    In CSP, *f* = number of violated constraints or *conflicts* (perhaps with weights). We stop when *f* = 0.
- **Iterative best improvement**: choose the successor that *most improves* the current state according to an evaluation function *f* (*greedy ascent/descent*). If more than one, choose at random.
  - ✓ *Iterative best improvement* algorithms move to the best successor, even if worse than current state. They may be stuck in a loop. Not complete.
  - ✓ Local optimum vs global optimum

Different search spaces



Visualization in one dimension, but in practice you have very high-dimensional spaces

a) a global random restart could be necessary to find the global minimum

b) a random walk combined with greedy descent enables it to escape local minima

Which of the methods works best is, typically, an empirical question

Stochastic local search

- **Stochastic local search algorithms** combine Iterative best improvement with randomness
- Randomness can be used to **escape local minima**:
  - ✓ **Random restart** is a global random move: the search start from a completely different part of the search space.
  - ✓ **Random walk** is local random move: random steps are taken interleaved with the optimizing steps.
- Algorithms differ in how much effort they put in computing the best improvement step.

Variants of stochastic local search

Which successor to select? Different strategies of different complexity …

- **Most improving step**

  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*. We need strategies for efficient computation.

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

- **Any conflict**
  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

For a more detailed description and evaluation of complexity see [Poole &Macworth]

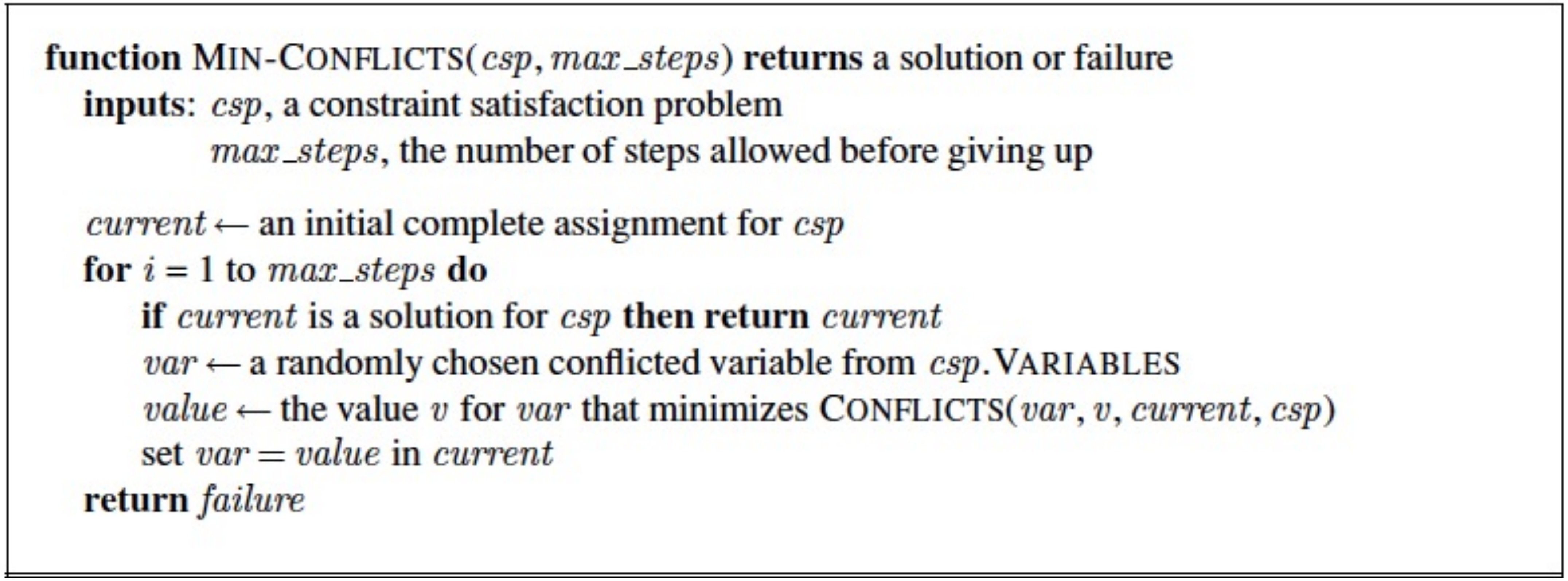
Min-conflict heuristics

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.

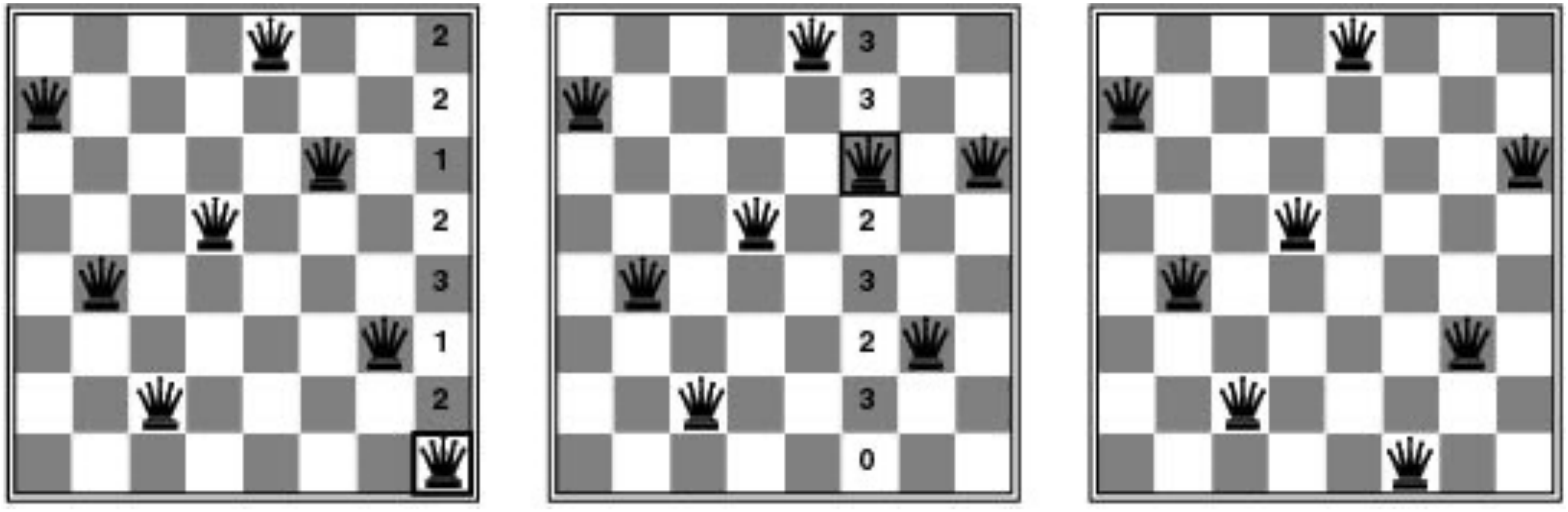
Min-conflicts algorithm



```
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 ← an initial complete assignment for csp
  for i = 1 to max_steps do
      if current is a solution for csp then return current
      var ← a randomly chosen conflicted variable from csp.VARIABLES
      value ← 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 instead 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. The parameter *t* is called the *tenure.*
- **Constraint weighting**:

  can help concentrate the search on the important constraints. We assign a numeric weight to each constraint. Initially 1, the weight is incremented each time the constraint is violated. The goal is to choose the variable and value which minimizes the weights of the violated constraints.

Local search: alternatives (see AIMA)

- **Simulated annealing**

  a technique for allowing downhill moves at the beginning of the algorithm and slowly *freezing* this possibility 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 (*fitness* score) 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.

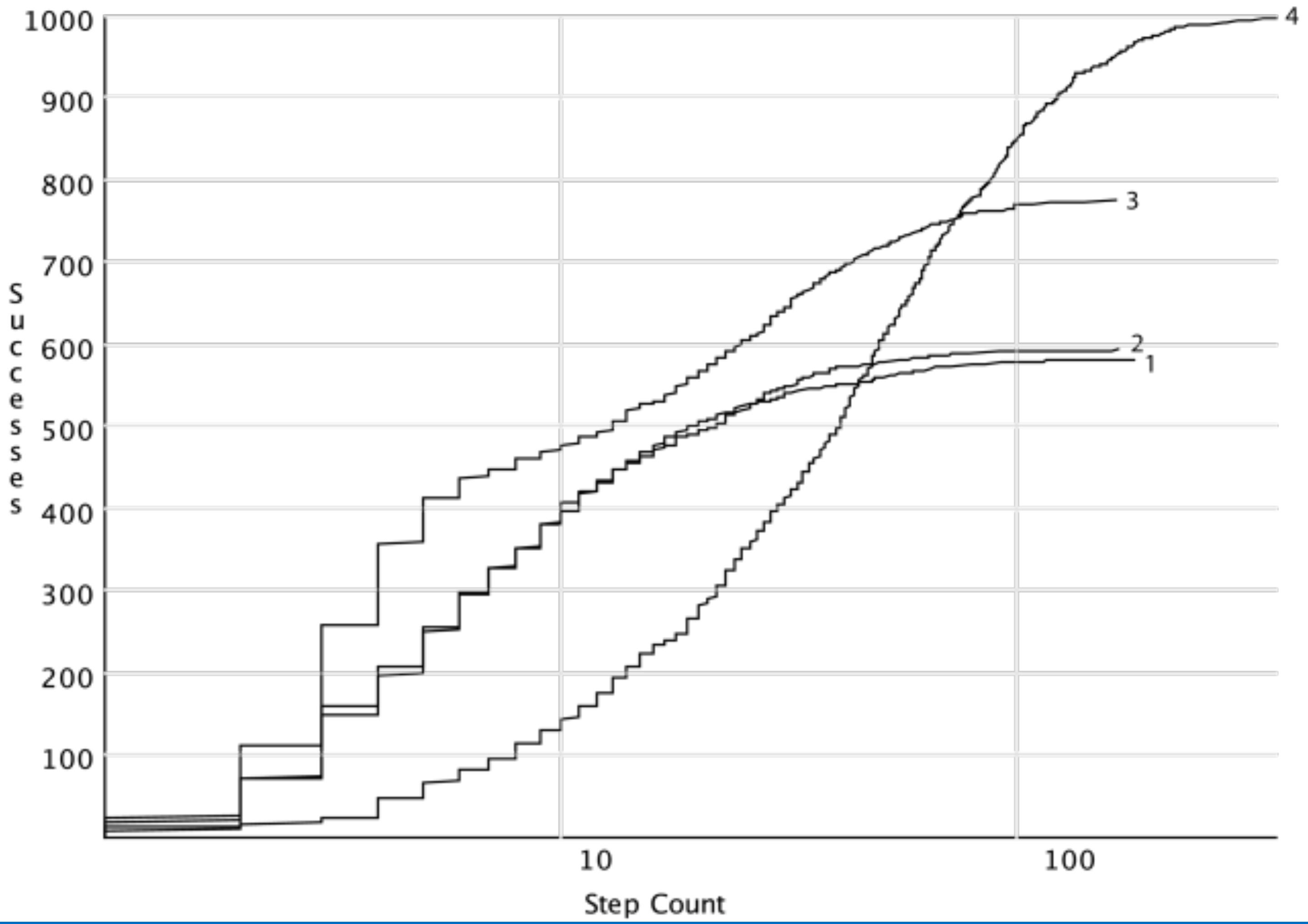
Consider complex flights schedules: a problem due to bad weather at one airport can render the schedule infeasible and require a rescheduling.

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 execution 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 compute the **run-time distribution**, which shows the number of runs in which the algorithm solved the problem within a given number of steps (or within a given run time).

Run time distribution [AI-FCA]



Four algorithms for the same instance of a problem

1000 runs for each algorithm

Logarithmic scale for the x-axis (limit to 1000 steps)

Algorithm 3 succeeds 80% of the times; algorithms 1 and 2, 60% of the times; algorithm 4, 100% of the times …

Algorithm 3 dominates 1 and 2.

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.

Conclusions

- We discussed a specialization of search algorithms for CSP
- Improvements in several directions are possible because of the specific representation of the state, and because constraints are exploited
  - ✓ Special heuristics for choosing variables and values
  - ✓ Problem reduction based on enforcing local consistency properties on the constraint graph
  - ✓ The idea of intelligent backtracking
- A review of local search method, and how they can be used for CSP
- Next: how we can semplify problems looking at their structure.