

# Lecture 6: Search 5

# General Solution Space Search

## & CSP

♂ C2b

# Constraint satisfaction problems (CSPs)

# Constraint satisfaction problems (CSPs)

Standard search problem:

state is a “black box”—any old data structure  
that supports goal test, eval, successor

CSP:

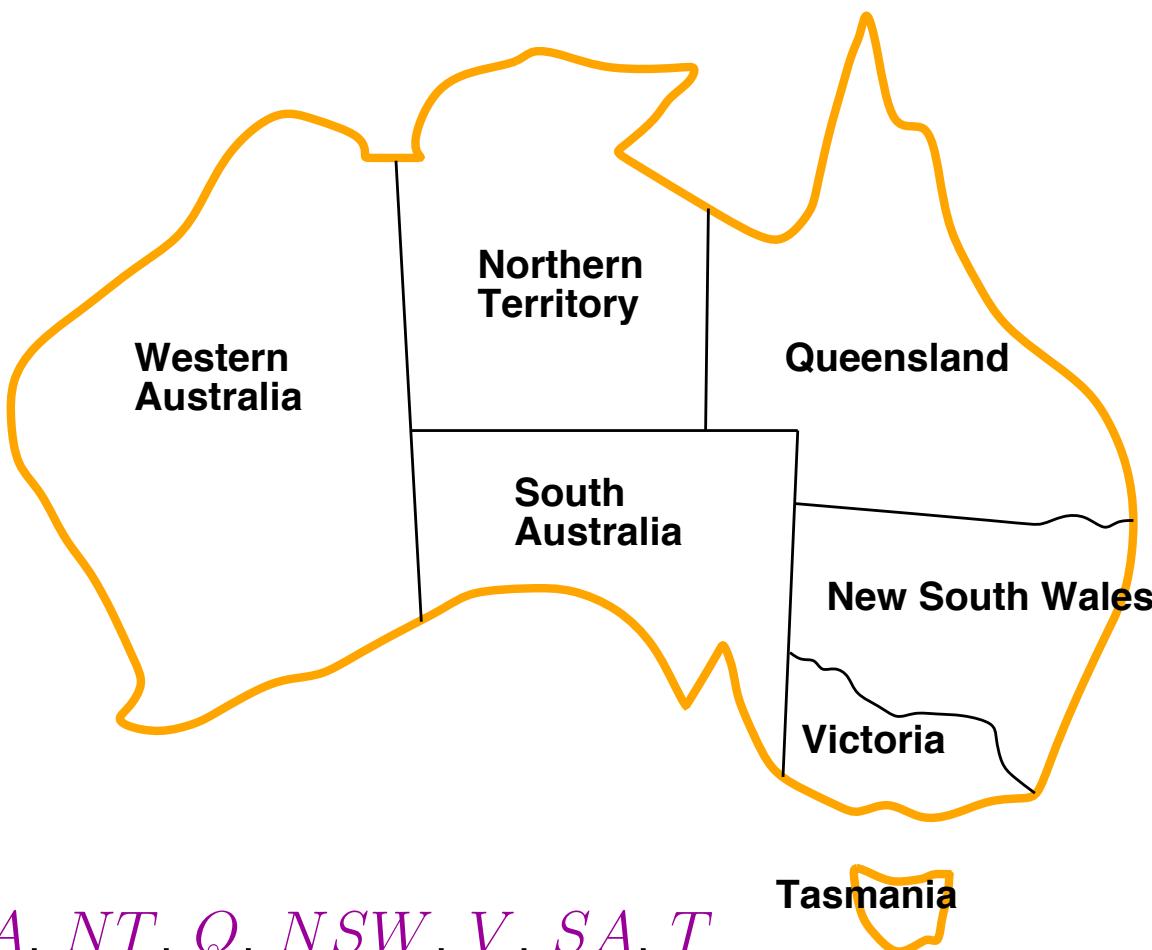
state is defined by variables  $X_i$  with values from domain  $D_i$

goal test is a set of constraints specifying  
allowable combinations of values for subsets of variables

Simple example of a formal representation language

Allows useful general-purpose algorithms with more power  
than standard search algorithms

# Example: Map-Coloring



Variables  $WA, NT, Q, NSW, V, SA, T$

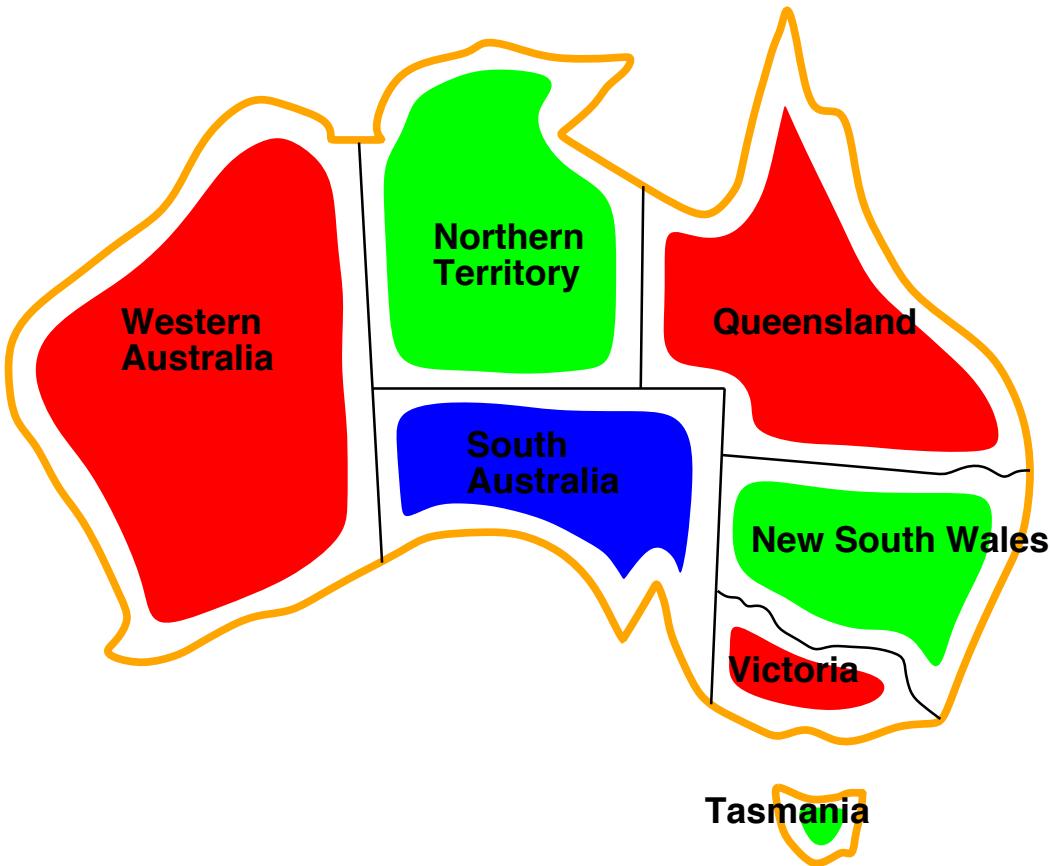
Domains  $D_i = \{red, green, blue\}$

Constraints: adjacent regions must have different colors

e.g.,  $WA \neq NT$  (if the language allows this), or

$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \dots\}$

# Example: Map-Coloring



Solutions are assignments satisfying all constraints, e.g.,

$$\{WA = \text{red}, NT = \text{green}, Q = \text{red}, NSW = \text{green}, V = \text{red}, SA = \text{blue}, T = \text{green}\}$$

# Standard search formulation (incremental)

NJU AI

Let's start with the straightforward, dumb approach, then fix it

States are 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.  
⇒ fail if no legal assignments (not fixable!)
- ◊ **Goal test:** the current assignment is complete

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

# Backtracking search

Variable assignments are **commutative**, i.e.,

$[WA = \text{red} \text{ then } NT = \text{green}]$  same as  $[NT = \text{green} \text{ then } WA = \text{red}]$

Only need to consider assignments to a single variable at each node

$\Rightarrow b = d$  and there are  $d^n$  leaves

Depth-first search for CSPs with single-variable assignments  
is called **backtracking** search

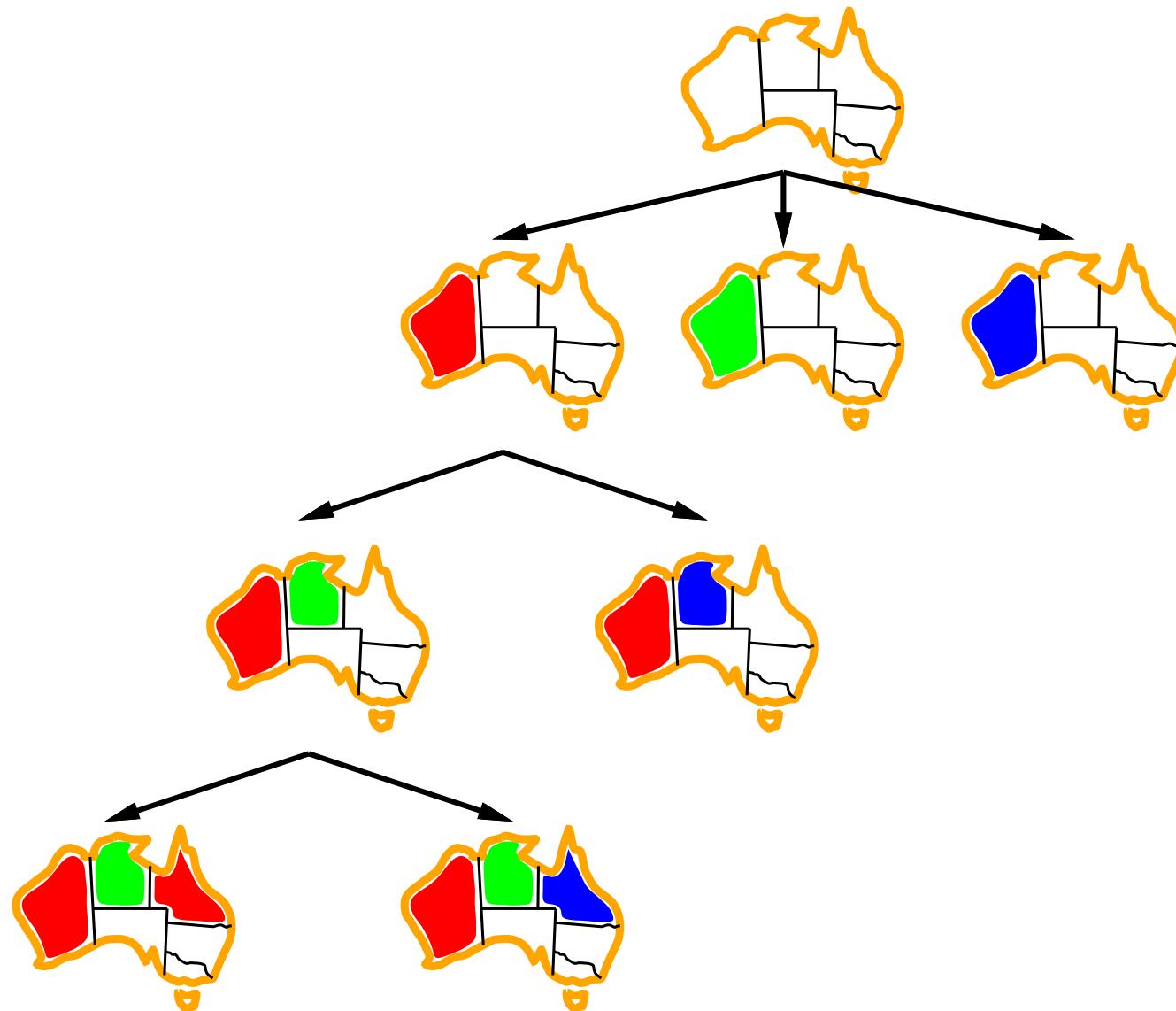
Backtracking search is the basic uninformed algorithm for CSPs

Can solve  $n$ -queens for  $n \approx 25$

# Backtracking search

```
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  $\leftarrow$  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  $\leftarrow$  RECURSIVE-BACKTRACKING(assignment, csp)
      if result  $\neq$  failure then return result
      remove {var = value} from assignment
  return failure
```

# Backtracking search example



# Improving backtracking efficiency

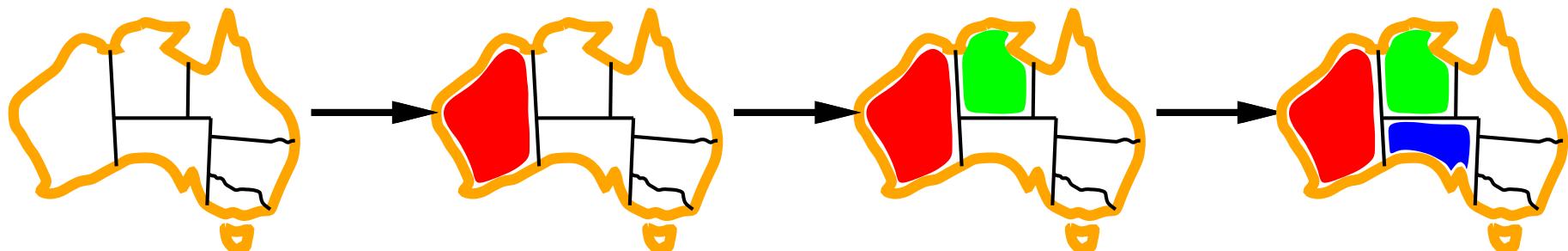
backtracking is uninformed  
make it more informed

**General-purpose** methods can give huge gains in speed:

1. Which variable should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early?
4. Can we take advantage of problem structure?

# Minimum remaining values

Minimum remaining values (MRV):  
choose the variable with the fewest legal values

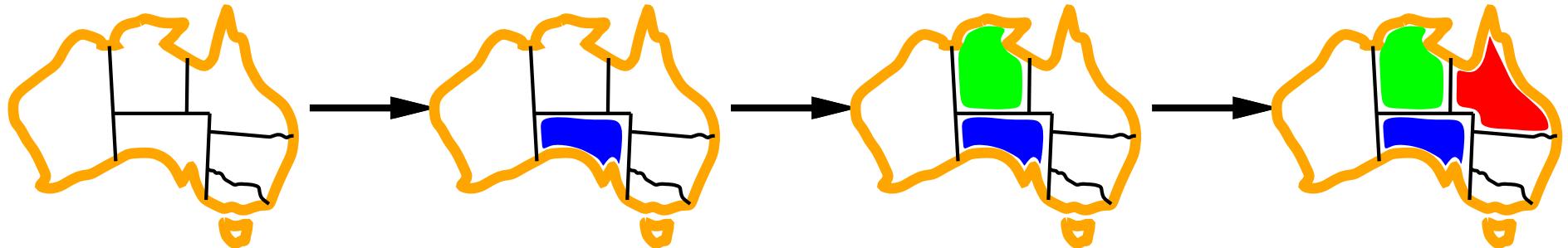


# Degree heuristic

Tie-breaker among MRV variables

Degree heuristic:

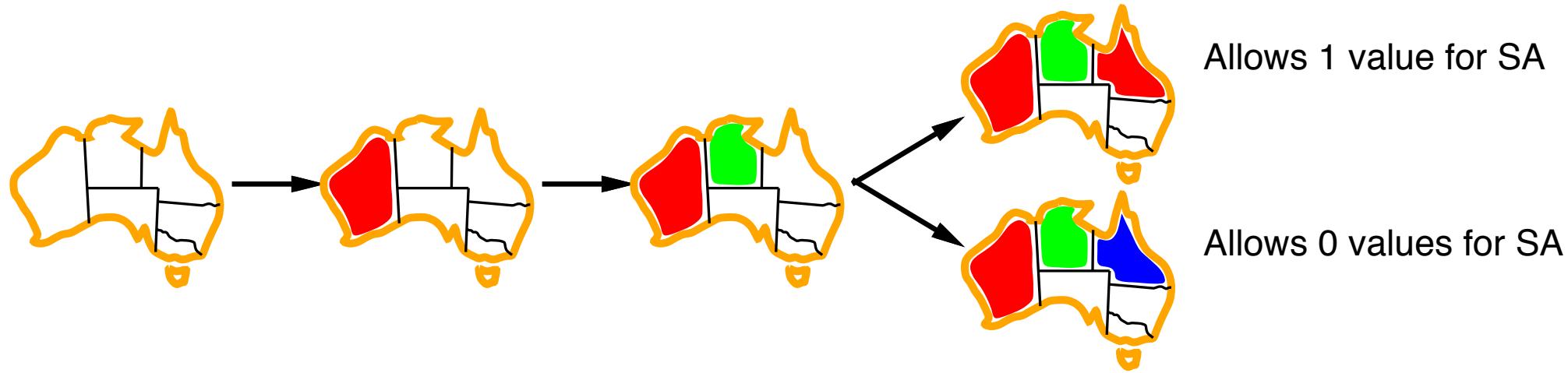
choose the variable with the most constraints on remaining variables



# Least constraining value

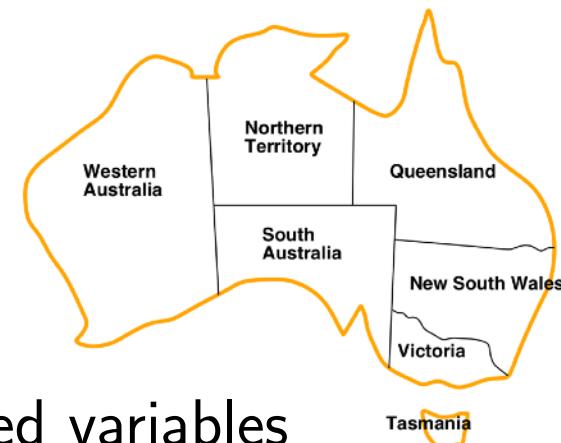


Given a variable, choose the least constraining value:  
the one that rules out the fewest values in the remaining variables

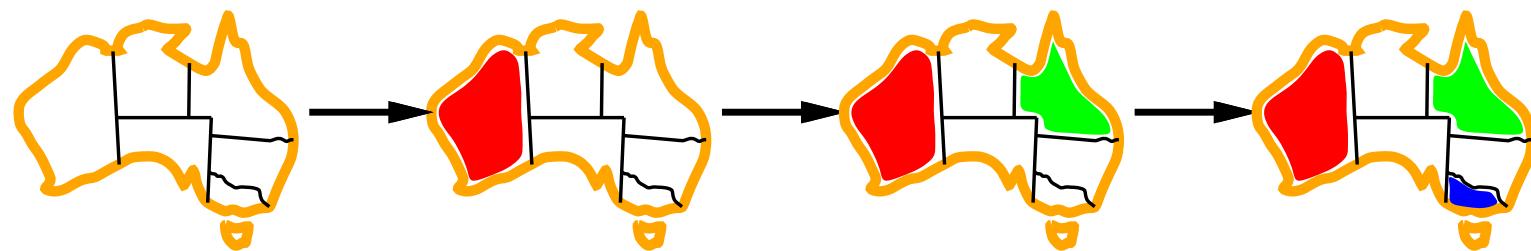


Combining these heuristics makes 1000 queens feasible

# Forward checking



Idea: Keep track of remaining legal values for unassigned variables  
Terminate search when any variable has no legal values



WA	NT	Q	NSW	V	SA	T
[Red, Green, Blue]						
[Red]		[Red, Green, Blue]	[Red, Green, Blue]	[Red, Green, Blue]		[Red, Green, Blue]
[Red]		[Blue]	[Green]	[Red, Blue]		[Red, Green, Blue]
[Red]		[Green]	[Red]	[Blue]		[Red, Green, Blue]

# Constraint propagation



Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



WA	NT	Q	NSW	V	SA	T
█	█	█	█	█	█	█
█		█	█	█	█	█
█			█	█	█	█

*NT* and *SA* cannot both be blue!

Constraint propagation repeatedly enforces constraints locally

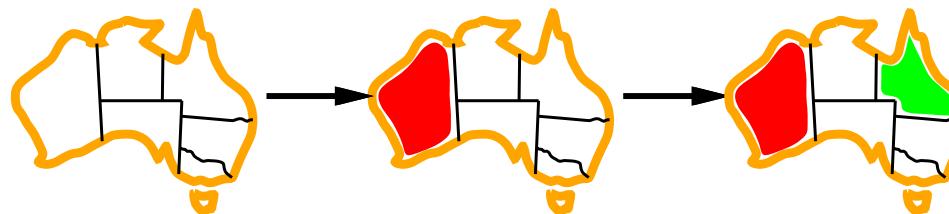
# Arc consistency



Simplest form of propagation makes each arc consistent

$X \rightarrow Y$  is consistent iff

for **every** value  $x$  of  $X$  there is **some** allowed  $y$



WA

NT

Q

NSW

V

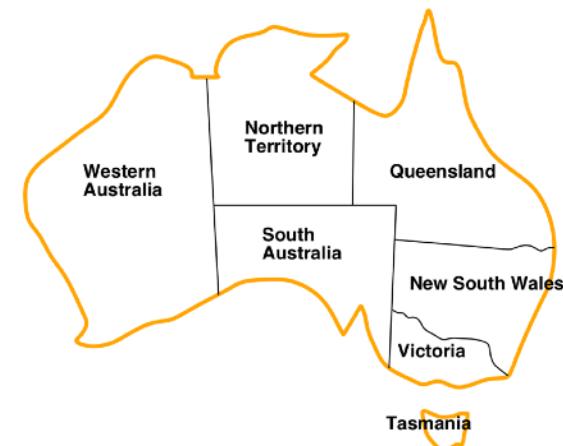
SA

T

--	--	--	--	--	--	--	--	--	--	--



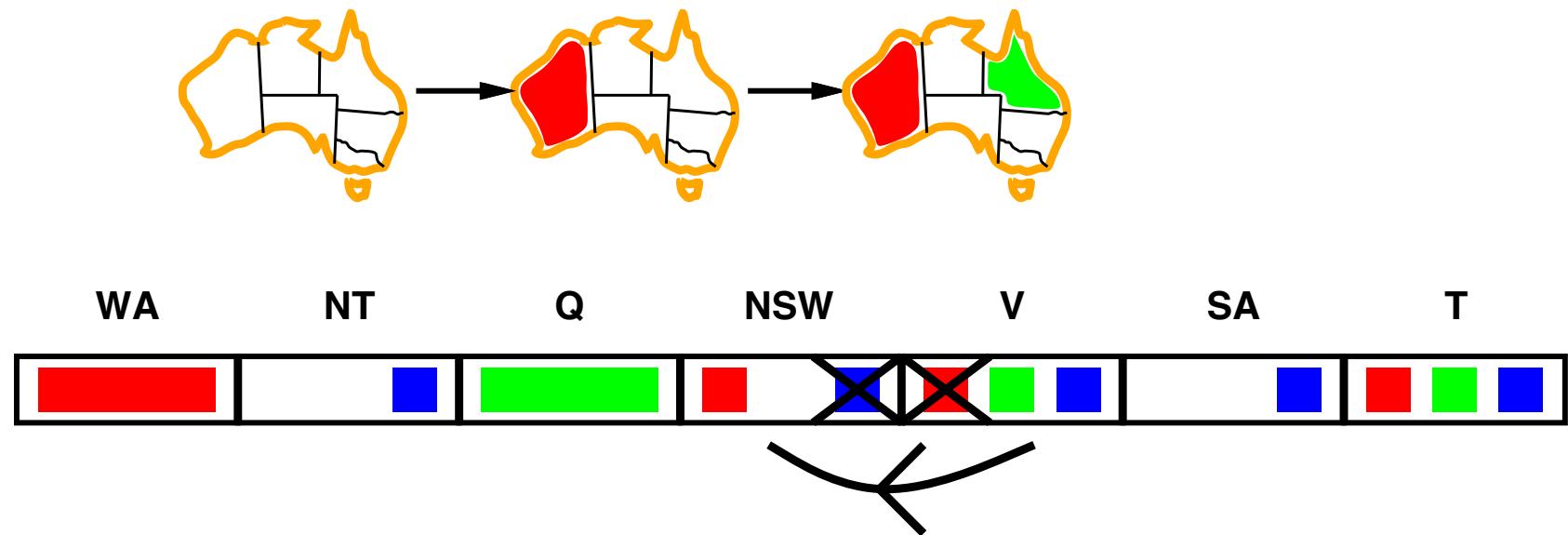
# Arc consistency



Simplest form of propagation makes each arc **consistent**

$X \rightarrow Y$  is consistent iff

for **every** value  $x$  of  $X$  there is **some** allowed  $y$



If  $X$  loses a value, neighbors of  $X$  need to be rechecked

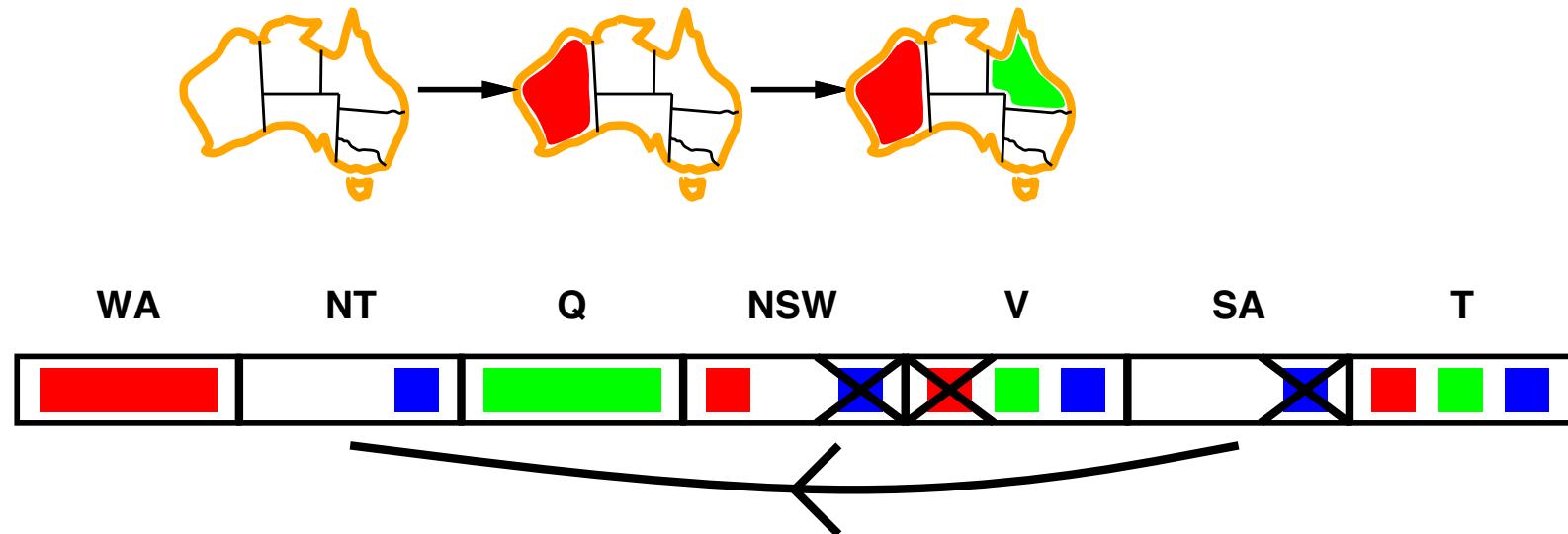
# Arc consistency



Simplest form of propagation makes each arc consistent

$X \rightarrow Y$  is consistent iff

for **every** value  $x$  of  $X$  there is **some** allowed  $y$



If  $X$  loses a value, neighbors of  $X$  need to be rechecked

Arc consistency detects failure earlier than forward checking

Can be run as a preprocessor or after each assignment

# Arc consistency

```

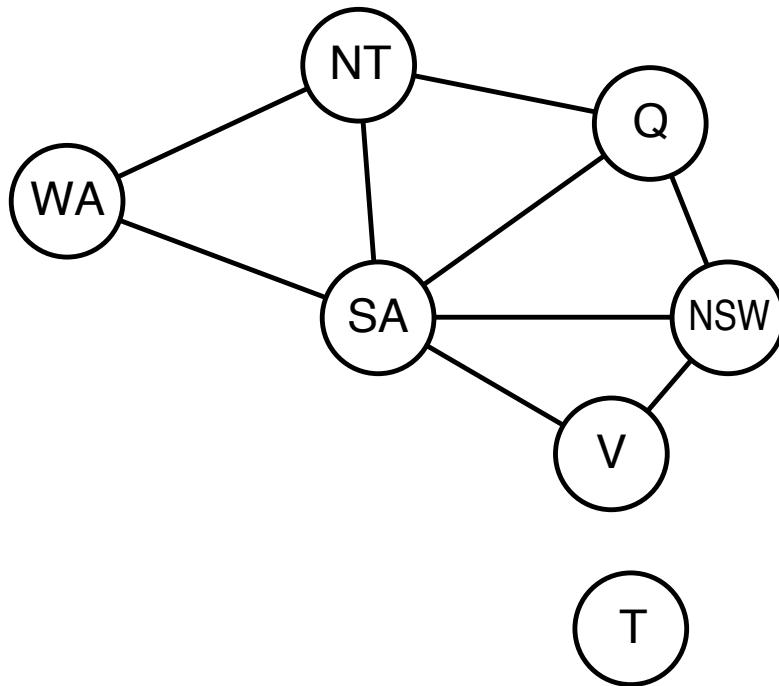
function AC-3(csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables  $\{X_1, X_2, \dots, X_n\}$ 
  local variables: queue, a queue of arcs, initially all the arcs in csp
  while queue is not empty do
     $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\text{queue})$ 
    if REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) then
      for each  $X_k$  in NEIGHBORS[ $X_i$ ] do
        add  $(X_k, X_i)$  to queue

function REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) returns true iff succeeds
  removed  $\leftarrow \text{false}$ 
  for each  $x$  in DOMAIN[ $X_i$ ] do
    if no value  $y$  in DOMAIN[ $X_j$ ] allows  $(x, y)$  to satisfy the constraint  $X_i \leftrightarrow X_j$ 
      then delete  $x$  from DOMAIN[ $X_i$ ]; removed  $\leftarrow \text{true}$ 
  return removed

```

$O(n^2d^3)$ , can be reduced to  $O(n^2d^2)$  (but detecting **all** is NP-hard)

# Problem Structure



Tasmania and mainland are **independent subproblems**

Identifiable as **connected components** of constraint graph

Suppose each subproblem has  $c$  variables out of  $n$  total

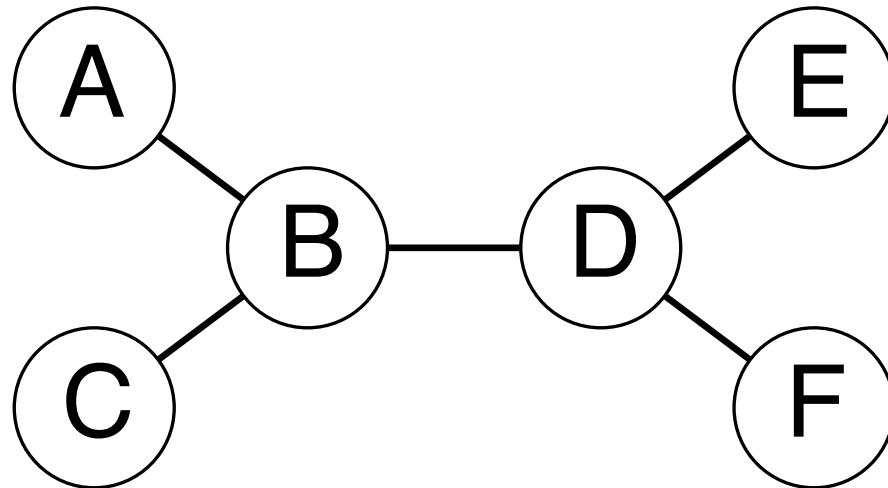
Worst-case solution cost is  $n/c \cdot d^c$ , **linear** in  $n$

E.g.,  $n = 80$ ,  $d = 2$ ,  $c = 20$

$2^{80} = 4$  billion years at 10 million nodes/sec

$4 \cdot 2^{20} = 0.4$  seconds at 10 million nodes/sec

# Tree-structured CSPs



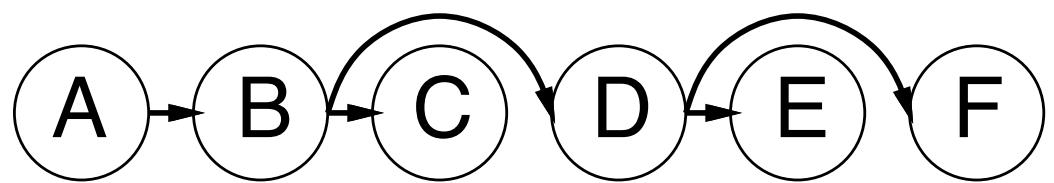
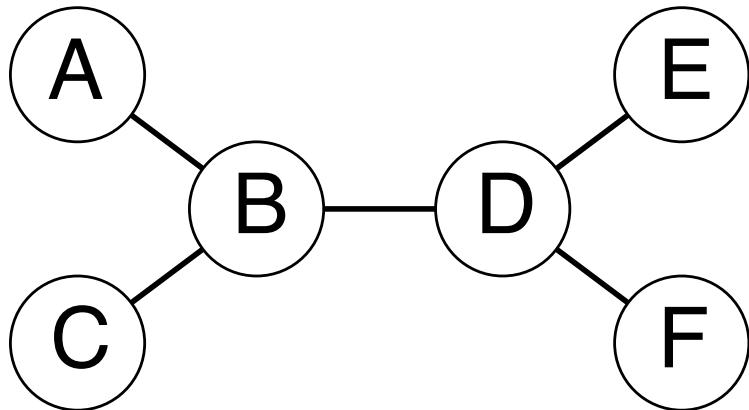
Theorem: if the constraint graph has no loops, the CSP can be solved in  $O(nd^2)$  time

Compare to general CSPs, where worst-case time is  $O(d^n)$

This property also applies to logical and probabilistic reasoning:  
an important example of the relation between syntactic restrictions  
and the complexity of reasoning.

# Algorithm for tree-structured CSPs

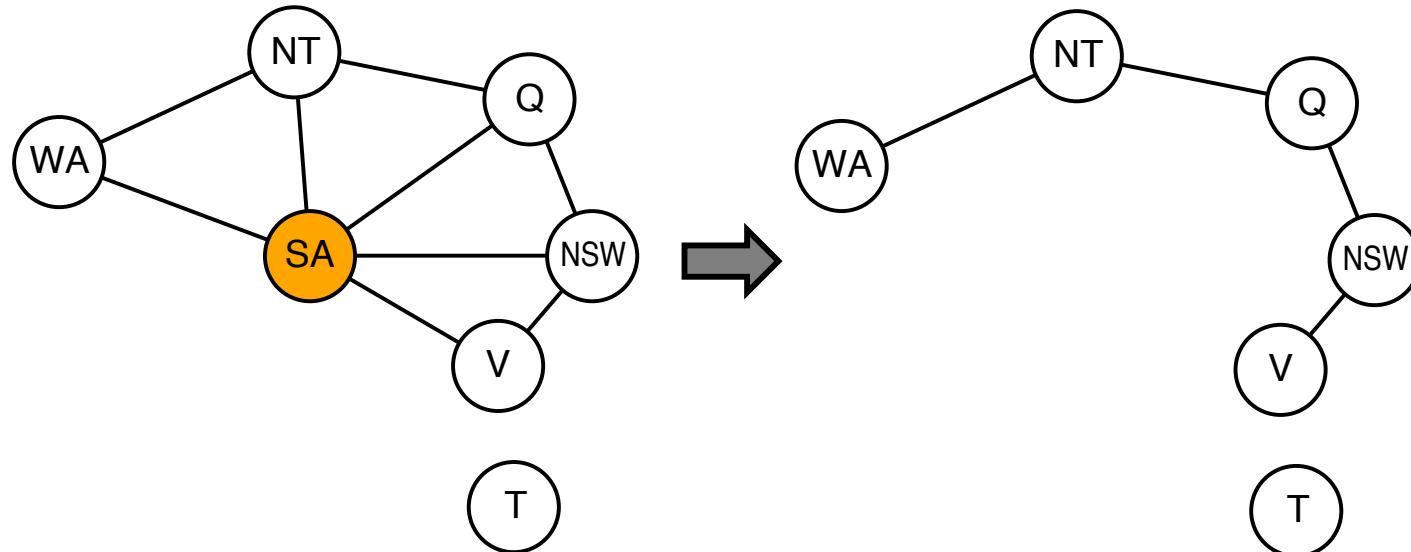
1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering



2. For  $j$  from  $n$  down to 2, apply  $\text{REMOVEINCONSISTENT}(\text{Parent}(X_j), X_j)$
3. For  $j$  from 1 to  $n$ , assign  $X_j$  consistently with  $\text{Parent}(X_j)$

# Nearly tree-structured CSPs

Conditioning: instantiate a variable, prune its neighbors' domains



Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

Cutset size  $c \Rightarrow$  runtime  $O(d^c \cdot (n - c)d^2)$ , very fast for small  $c$

# Iterative algorithms for CSPs

Hill-climbing, simulated annealing typically work with “complete” states, i.e., all variables assigned

To apply to CSPs:

- allow states with unsatisfied constraints
- operators **reassign** variable values

Variable selection: randomly select any conflicted variable

Value selection by **min-conflicts** heuristic:

- choose value that violates the fewest constraints
- i.e., hillclimb with  $h(n)$  = total number of violated constraints

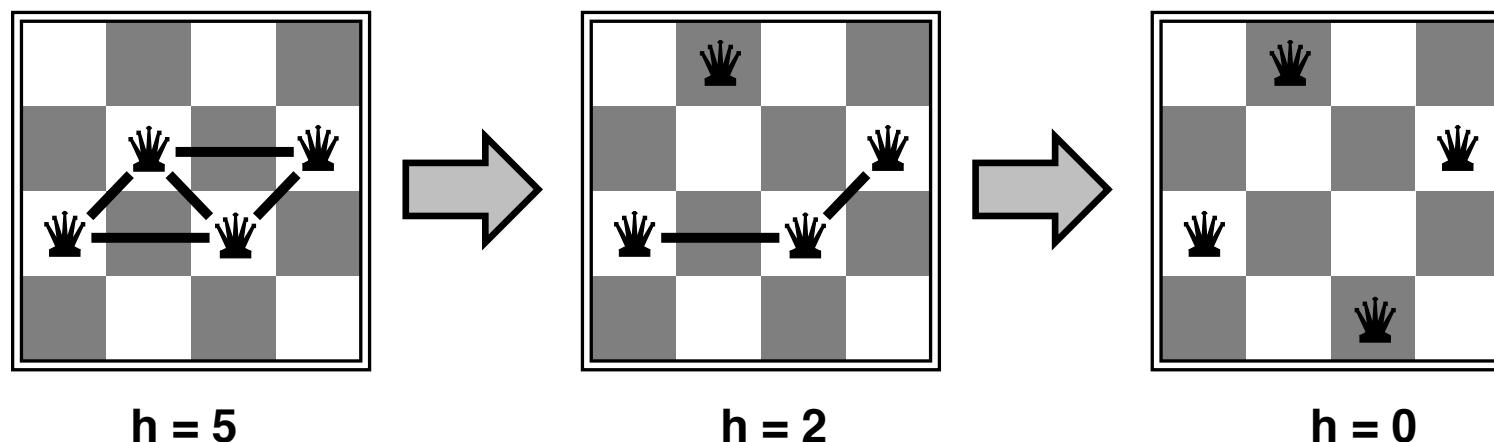
# Example: 4-Queens

States: 4 queens in 4 columns ( $4^4 = 256$  states)

Operators: move queen in column

Goal test: no attacks

Evaluation:  $h(n)$  = number of attacks

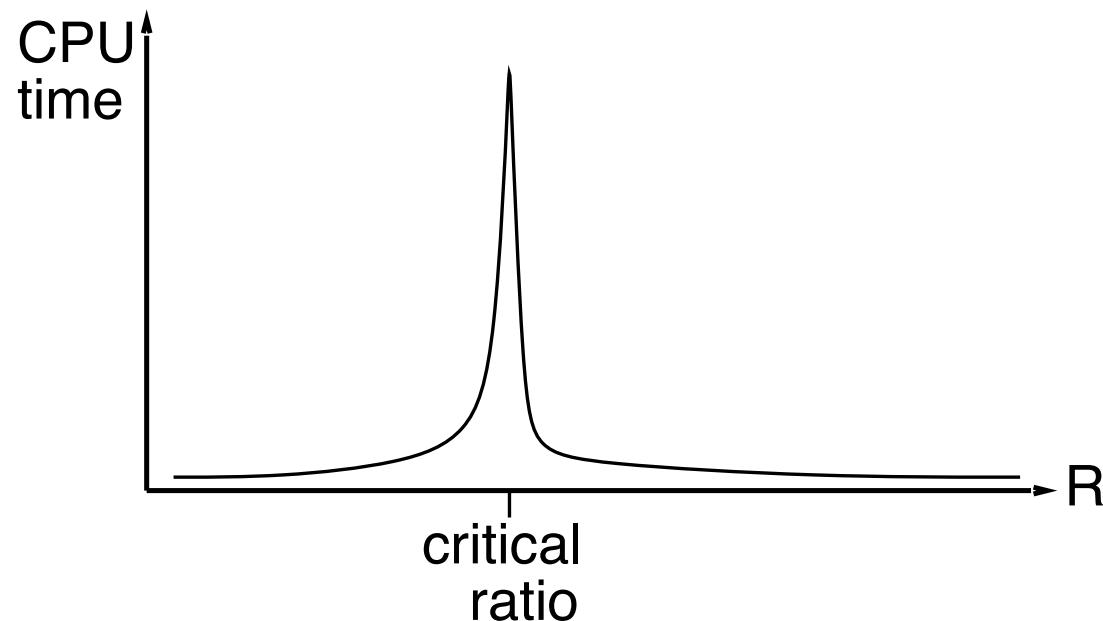


# Performance of min-conflicts

Given random initial state, can solve  $n$ -queens in almost constant time for arbitrary  $n$  with high probability (e.g.,  $n = 10,000,000$ )

The same appears to be true for any randomly-generated CSP  
**except** in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$



# Varieties of CSPs

## Discrete variables

finite domains; size  $d$   $\Rightarrow O(d^n)$  complete assignments

- ◊ e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)

infinite domains (integers, strings, etc.)

- ◊ e.g., job scheduling, variables are start/end days for each job
- ◊ need a **constraint language**, e.g.,  $StartJob_1 + 5 \leq StartJob_3$
- ◊ linear constraints solvable, nonlinear undecidable

## Continuous variables

- ◊ e.g., start/end times for Hubble Telescope observations
- ◊ linear constraints solvable in poly time by LP methods

# Varieties of CSPs

Unary constraints involve a single variable,  
e.g.,  $SA \neq green$

Binary constraints involve pairs of variables,  
e.g.,  $SA \neq WA$

Higher-order constraints involve 3 or more variables,  
e.g., cryptarithmetic column constraints

Preferences (soft constraints), e.g., *red* is better than *green*  
often representable by a cost for each variable assignment  
→ constrained optimization problems

# Real-world CSPs

Assignment problems

e.g., who teaches what class

Timetabling problems

e.g., which class is offered when and where?

Hardware configuration

Spreadsheets

Transportation scheduling

Factory scheduling

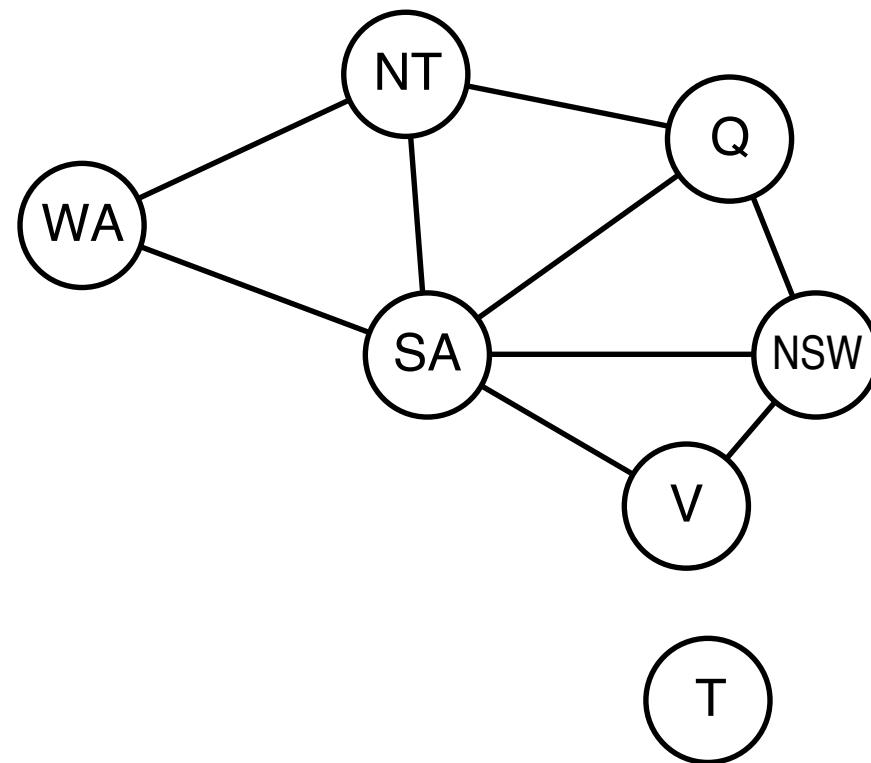
Floorplanning

Notice that many real-world problems involve real-valued variables

# Constraint graph

Binary CSP: each constraint relates at most two variables

Constraint graph: nodes are variables, arcs show constraints



General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!

# Convert higher-order to binary

A higher-order constraint can be converted to binary constraints with a *hidden-variable*

variable: A, B, C   domain: {1,2,3}   constraint:  $A+B=C$

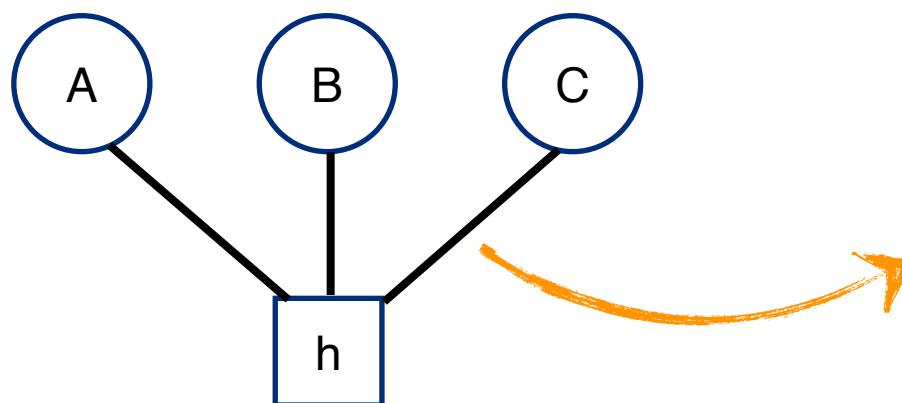
all possible assignments:  $(A,B,C) = (1,1,2), (1,2,3), (2,1,3)$

*hidden-variable*: h with

domain: {1,2,3}

(each value corresponds to  
an assignment)

the constraint graph:



constraint:  
 $h=1, C=2$   
 $h=2, C=3$   
 $h=3, C=3$

from the definition of h

# Example: Cryptarithmetic

$$\begin{array}{r}
 \text{T} \ \text{W} \ \text{O} \\
 + \ \text{T} \ \text{W} \ \text{O} \\
 \hline
 \text{F} \ \text{O} \ \text{U} \ \text{R}
 \end{array}$$

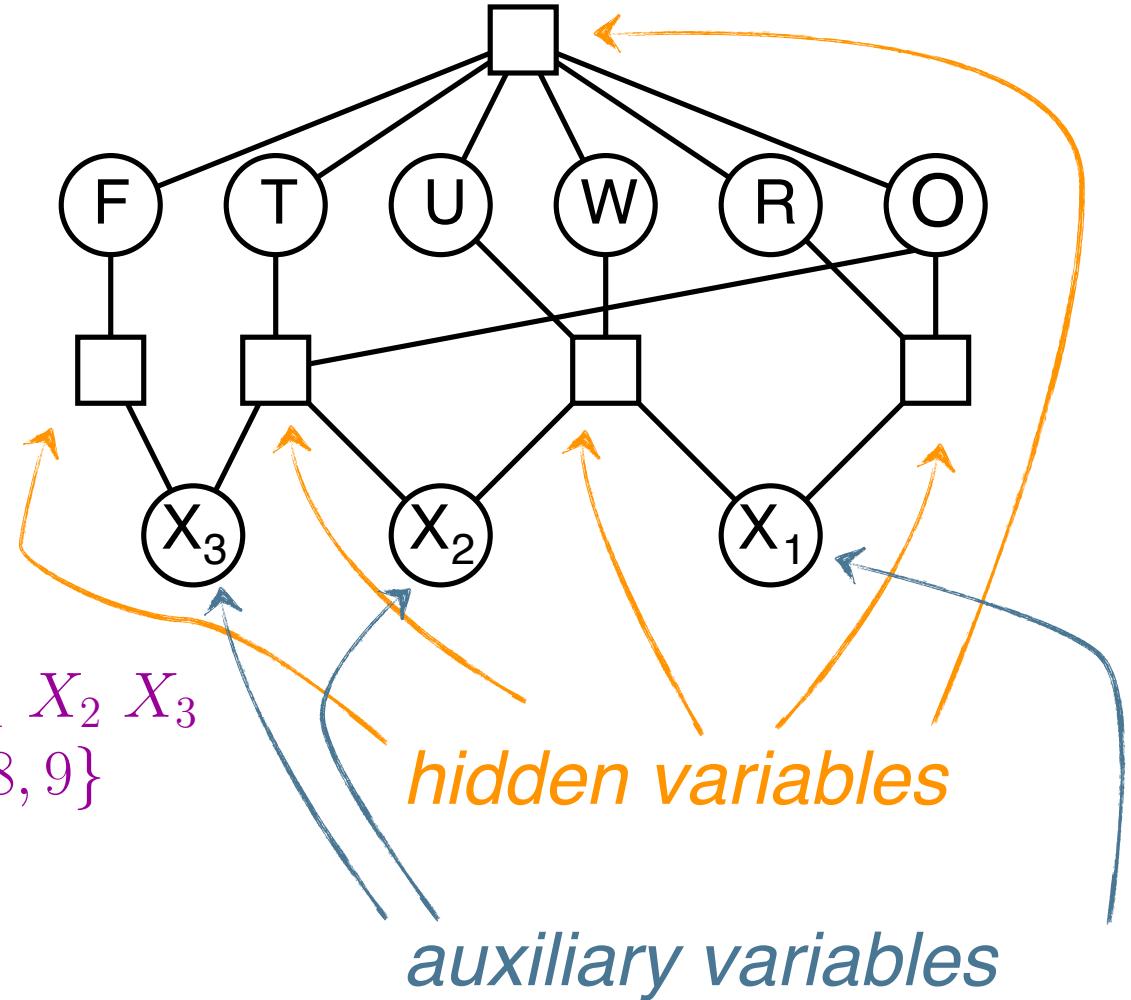
Variables:  $F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3$

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

Constraints

$\text{alldiff}(F, T, U, W, R, O)$

$O + O = R + 10 \cdot X_1$ , etc.



# Summary of CSP

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

Backtracking = depth-first search with one variable assigned per node

Variable ordering and value selection heuristics help significantly

Forward checking prevents assignments that guarantee later failure

Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies

The CSP representation allows analysis of problem structure

Tree-structured CSPs can be solved in linear time

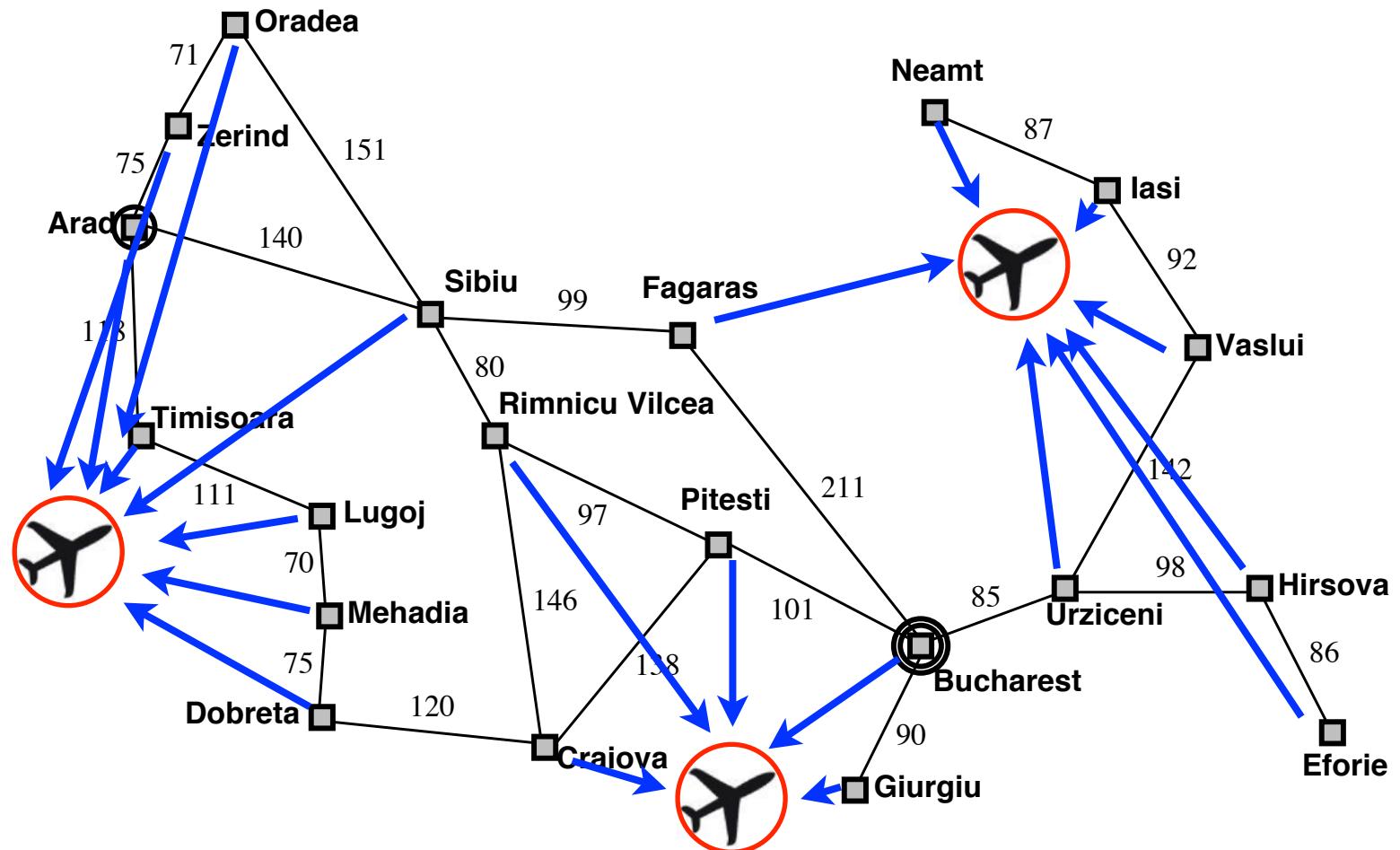
Iterative min-conflicts is usually effective in practice

# General (Iterative) Solution Space Search

# Greedy idea in continuous space

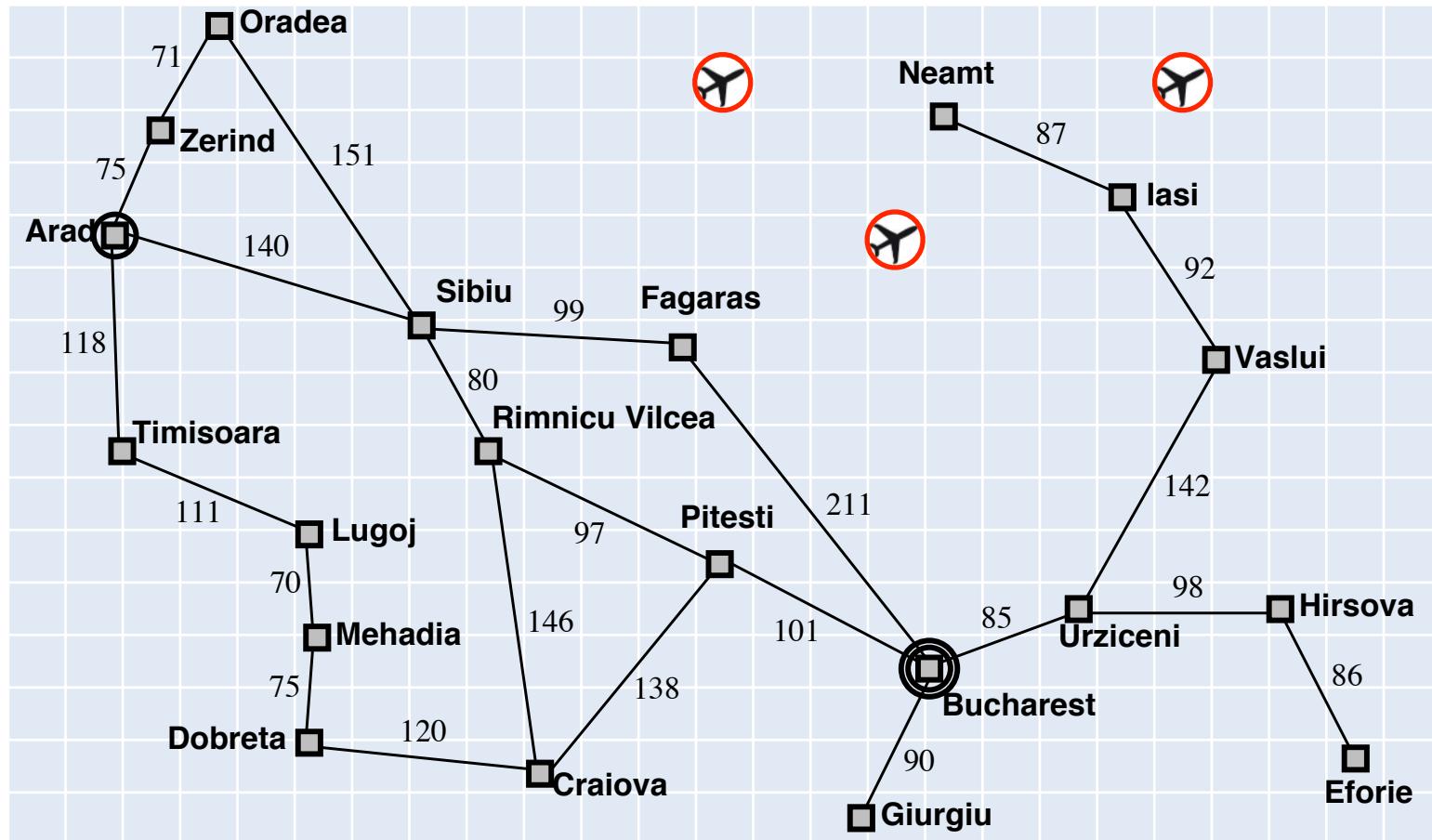
Suppose we want to site three airports in Romania:

- 6-D state space defined by  $(x_1, y_1), (x_2, y_2), (x_3, y_3)$
- objective function  $f(x_1, y_1, x_2, y_2, x_3, y_3) =$   
sum of squared distances from each city to nearest airport



# Greedy idea in continuous space

discretize and use hill climbing



# Hill climbing

```
function HillClimb_Step(double[] solution)
    double value = Eval(solution)
    List neighbors = Neighbors(solution)
    double bestv = value
    double[] bestc = none
    for each candidate in neighbors do
        double candivalue = eval(candidate)
        if candivalue < bestv then
            bestv = candivalue
            bestc = candidate
        end if
    end for
    return bestc
```

# Greedy idea in continuous space

## gradient decent

- 6-D state space defined by  $(x_1, y_2), (x_2, y_2), (x_3, y_3)$
- objective function  $f(x_1, y_2, x_2, y_2, x_3, y_3) =$   
sum of squared distances from each city to nearest airport

Gradient methods compute

$$\nabla f = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right)$$

to increase/reduce  $f$ , e.g., by  $\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$

*1-order method*

# Greedy idea in continuous space

## gradient decent

- 6-D state space defined by  $(x_1, y_2), (x_2, y_2), (x_3, y_3)$
- objective function  $f(x_1, y_2, x_2, y_2, x_3, y_3) =$   
sum of squared distances from each city to nearest airport

Sometimes can solve for  $\nabla f(\mathbf{x}) = 0$  exactly (e.g., with one city).  
 Newton–Raphson (1664, 1690) iterates  $\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_f^{-1}(\mathbf{x})\nabla f(\mathbf{x})$   
 to solve  $\nabla f(\mathbf{x}) = 0$ , where  $\mathbf{H}_{ij} = \partial^2 f / \partial x_i \partial x_j$

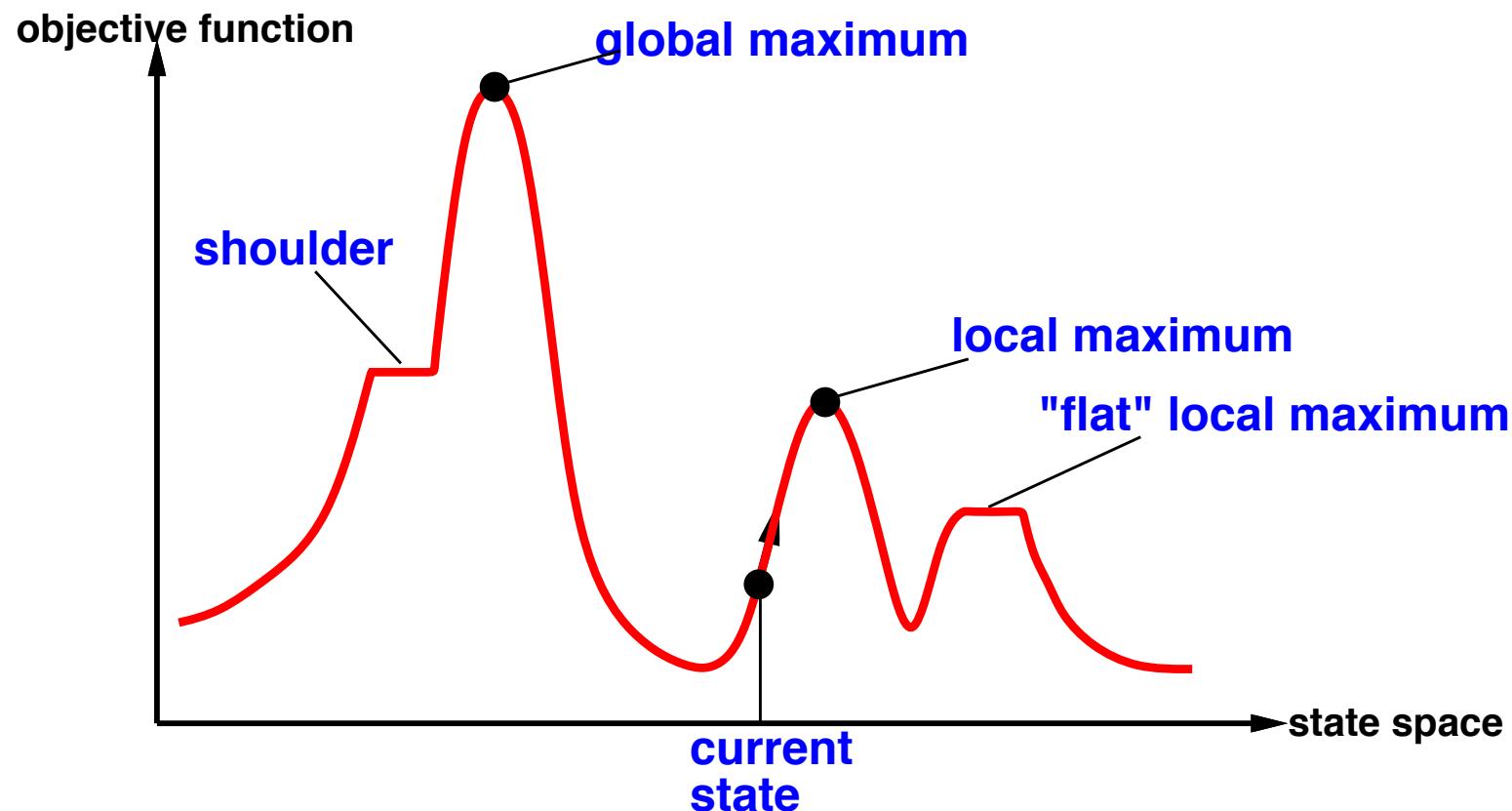
*2-order method*

Taylor's series:

$$f(x) = f(a) + (x - a)f'(a) + \frac{(x - a)^2}{2}f''(a) + \dots = \sum_{i=0}^{\infty} \frac{(x - a)^i}{i!} f^{(i)}(a).$$

# Greedy idea

1st and 2nd order methods may not find global optimal solutions  
they work for convex functions



# Purely random search

```
function RandomSearch_Step(double[] solution)
    double value = Eval(solution)
    double[] rsol = RandomSolution()
    double vr = Eval(rsol)
    if vr < value then
        return rsol
    end if
    return none
```

optimal after infinite steps! why?

can be more smart? replace RandomSolution

# Hill climbing vs. Pure random search

```
function HillClimb_Step(double[] solution)
    double value = Eval(solution)
    List neighbors = Neighbors(solution)
    double bestv = value
    double[] bestc = none
    for each candidate in neighbors do
        double candivalue = eval(candidate)
        if candivalue < bestv then
            bestv = candivalue
            bestc = candidate
        end if
    end for
    return bestc
```

```
function RandomSearch_Step(double[] solution)
    double value = Eval(solution)
    double[] rsol = RandomSolution()
    double vr = Eval(rsol)
    if vr < value then
        return rsol
    end if
    return none
```

exploitation vs. exploration  
locally optimal vs. globally optimal

# Meta-heuristics

“problem independent

“black-box

“zeroth-order method

...

and usually inspired from nature phenomenon

# Simulated annealing



temperature from high to low

when high temperature, form the shape

when low temperature, polish the detail

# Simulated annealing

Idea: escape local maxima by allowing some “bad” moves  
**but gradually decrease their size and frequency**

```

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  $\leftarrow$  MAKE-NODE(INITIAL-STATE[problem])
  for t  $\leftarrow$  1 to  $\infty$  do
    T  $\leftarrow$  schedule[t]
    if T = 0 then return current
    next  $\leftarrow$  a randomly selected successor of current
     $\Delta E \leftarrow \text{VALUE}[\text{next}] - \text{VALUE}[\text{current}]$ 
    if  $\Delta E > 0$  then current  $\leftarrow$  next
    else current  $\leftarrow$  next only with probability  $e^{\Delta E/T}$ 

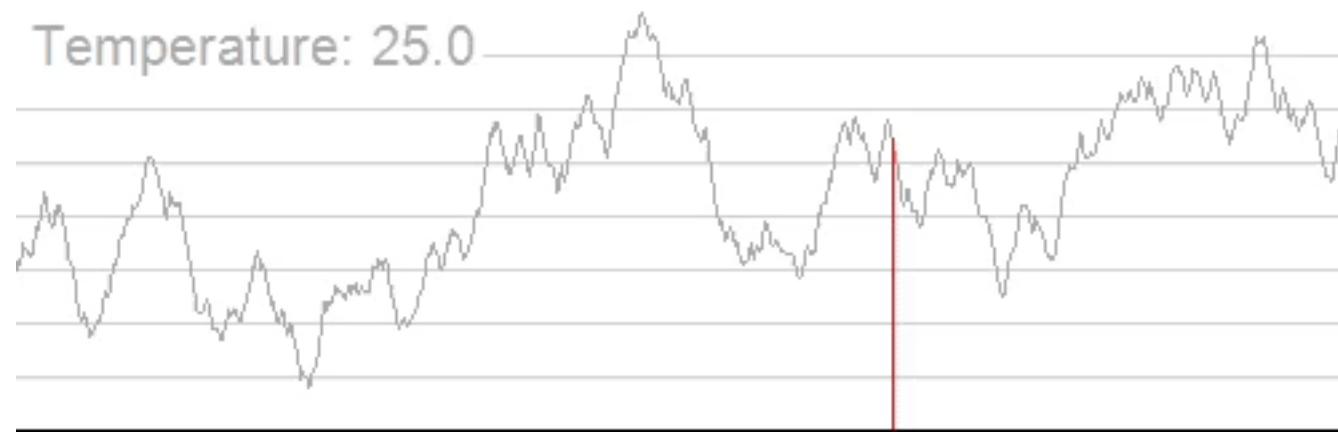
```

*the neighborhood range shrinks with T*

*the probability of accepting a bad solution decreases with T*

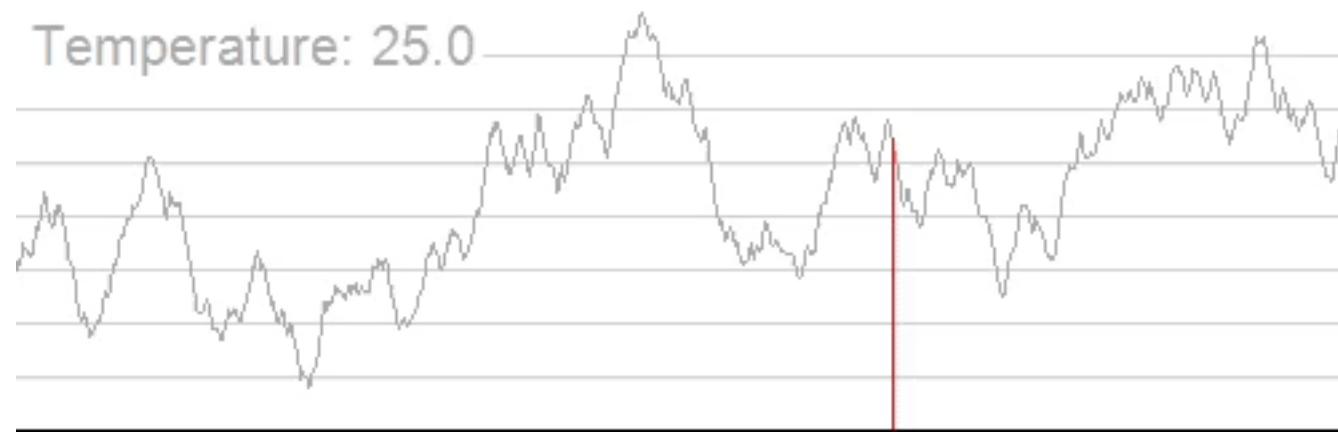
# Simulated annealing

a demo



# Simulated annealing

a demo



# Local beam search

Idea: keep  $k$  states instead of 1; choose top  $k$  of all their successors

Not the same as  $k$  searches run in parallel!

Searches that find good states recruit other searches to join them

Problem: quite often, all  $k$  states end up on same local hill

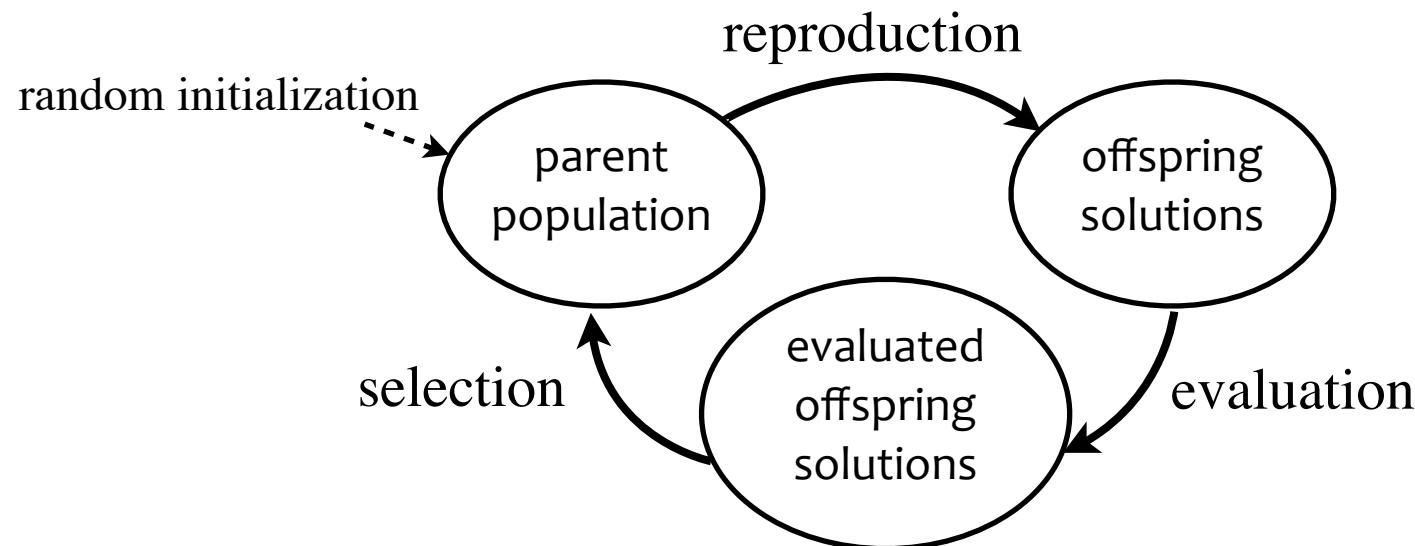
Idea: choose  $k$  successors randomly, biased towards good ones

Observe the close analogy to natural selection!

# Genetic algorithm

a simulation of Darwin's evolutionary theory  
(more generally: evolutionary algorithm)

- over-reproduction with diversity
- nature selection



# Genetic algorithm

Encode a solution as a vector,

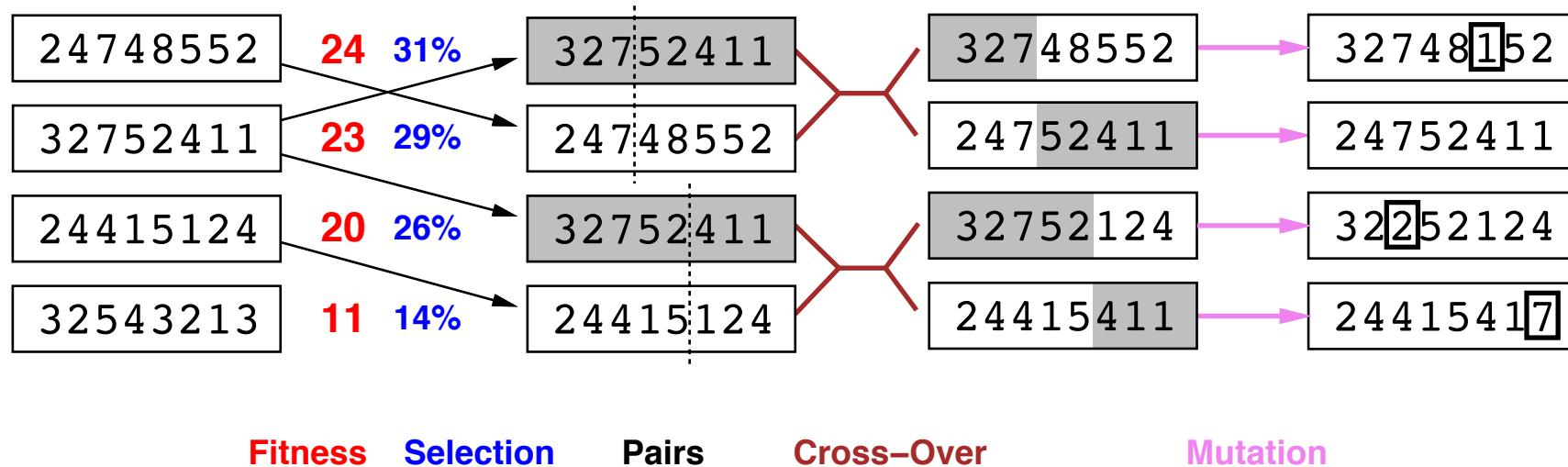
- 1:  $Pop \leftarrow n$  randomly drawn solutions from  $\mathcal{X}$
- 2: **for**  $t=1,2,\dots$  **do**
- 3:    $Pop^m \leftarrow \{\text{mutate}(s) \mid \forall s \in Pop\}$ , the mutated solutions
- 4:    $Pop^c \leftarrow \{\text{crossover}(s_1, s_2) \mid \exists s_1, s_2 \in Pop^m\}$ , the recombined solutions
- 5:   evaluate every solution in  $Pop^c$  by  $f(s)$  ( $\forall s \in Pop^c$ )
- 6:    $Pop^s \leftarrow$  selected solutions from  $Pop$  and  $Pop^c$
- 7:    $Pop \leftarrow Pop^s$
- 8:   **terminate** if meets a stopping criterion
- 9: **end for**

mutation: some kind of random changes

crossover: some kind of random exchanges

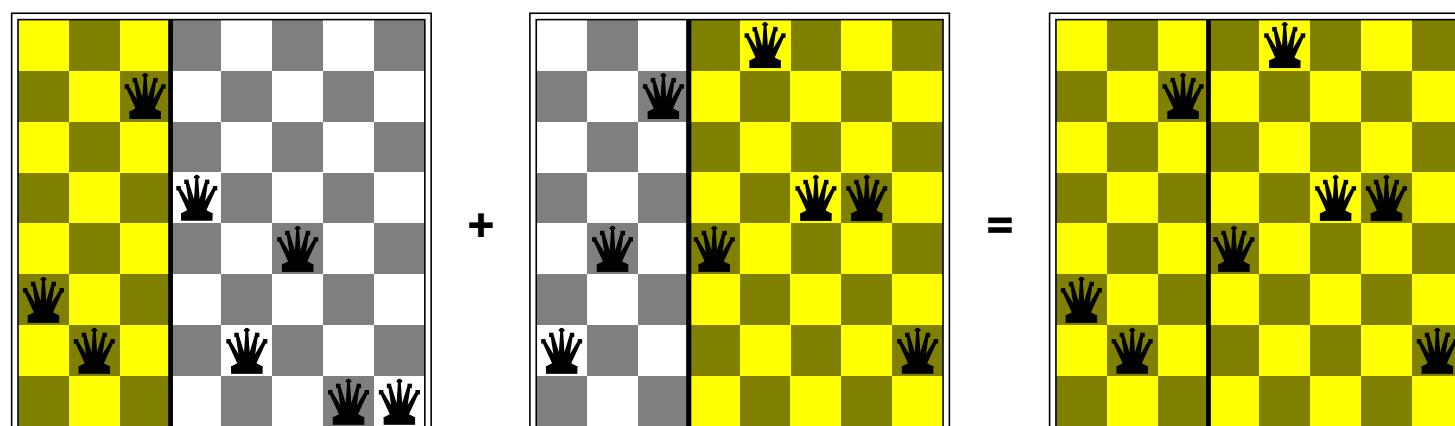
selection: some kind of quality related selection

# Genetic algorithm

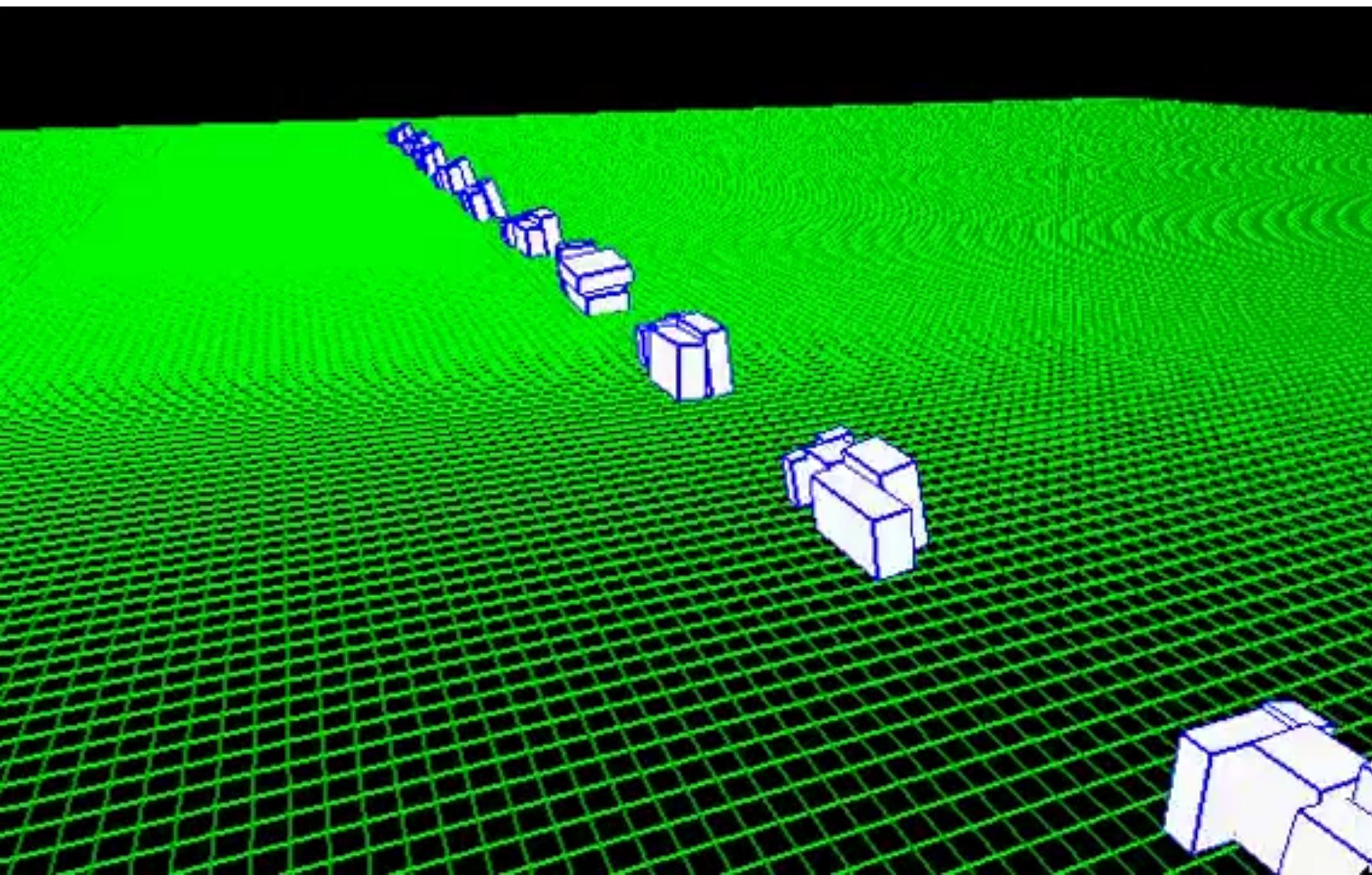


GAs require states encoded as strings (GPs use programs)

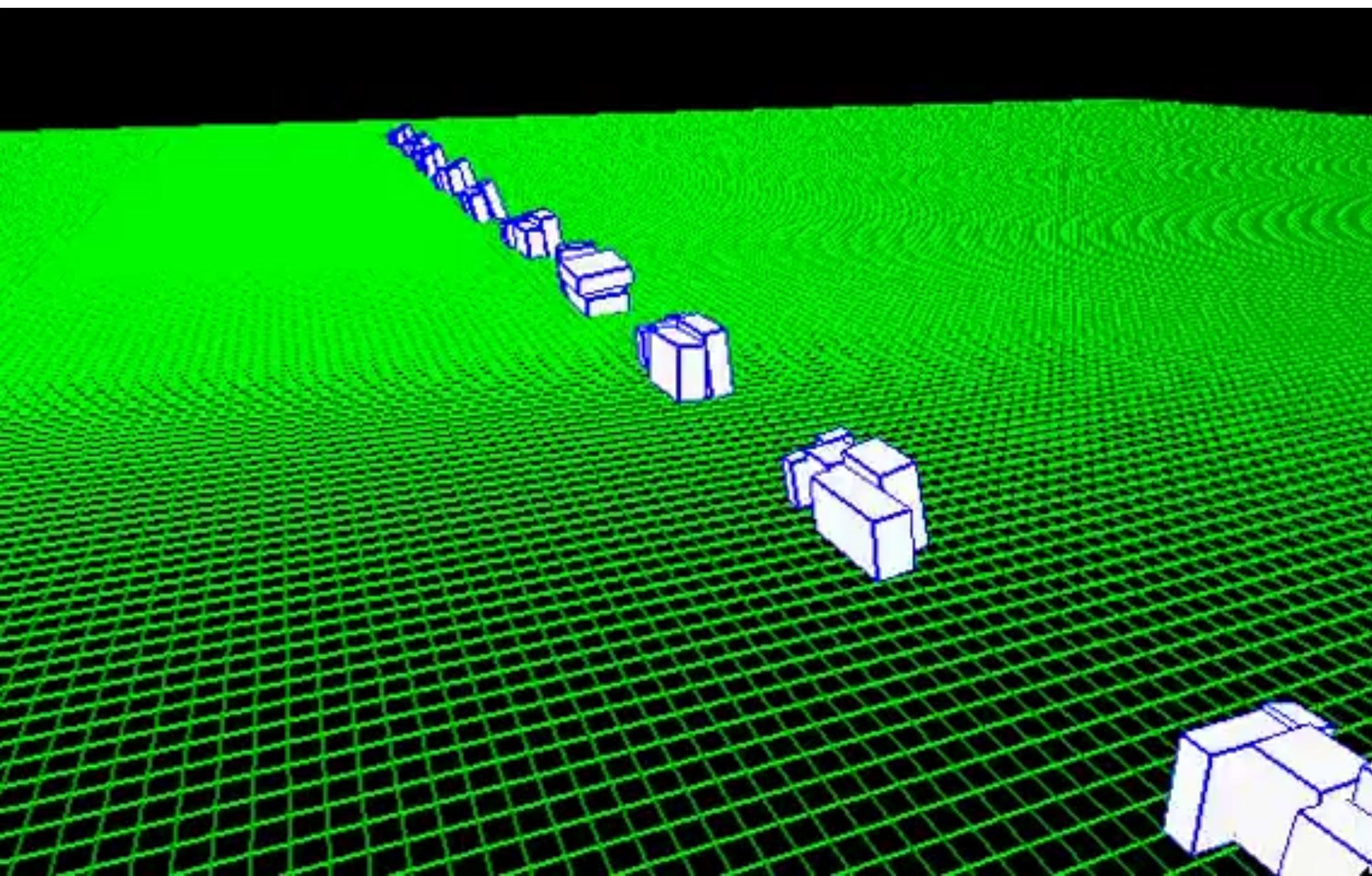
Crossover helps iff substrings are meaningful components



# An evolutionary of virtual life



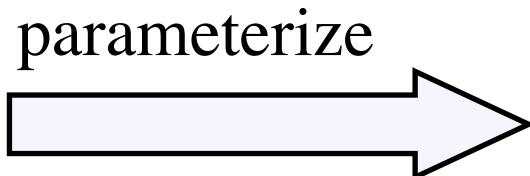
# An evolutionary of virtual life



# Example

*hard to apply traditional optimization methods  
but easy to test a given solution*

Representation:

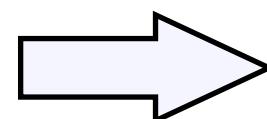
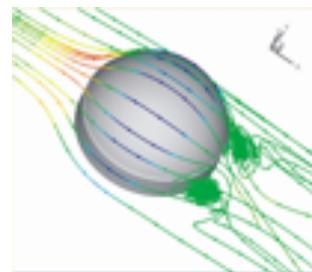
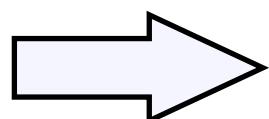


represented as a vector of parameters

Fitness:



$x_i$



$f(x_i)$

test by simulation/experiment

# Example



Series 700



Series N700

## Technological overview of the next generation Shinkansen high-speed train Series N700

M. Ueno<sup>1</sup>, S. Usui<sup>1</sup>, H. Tanaka<sup>1</sup>, A. Watanabe<sup>2</sup>

<sup>1</sup>*Central Japan Railway Company, Tokyo, Japan*, <sup>2</sup>*West Japan Railway Company, Osaka, Japan*

### Abstract

In March 2005, Central Japan Railway Company (JR Central) has completed prototype development of the Series N700, the next generation Shinkansen high-speed train. This paper

waves and other issues related to environmental compatibility such as external noise. To combat this, an aero double-wing-type has been adopted for nose shape (Fig. 3). This nose shape, which boasts the most appropriate aerodynamic performance, has been newly developed for railway rolling stock using the latest analytical technique (i.e. genetic algorithms) used to develop the main wings of airplanes. The shape resembles a bird in flight, suggesting a feeling of boldness and speed.

On the Tokaido Shinkansen line, Series N700 cars save 19% energy than Series 700 cars, despite a 30% increase in the output of their traction equipment for higher-speed operation (Fig. 4).

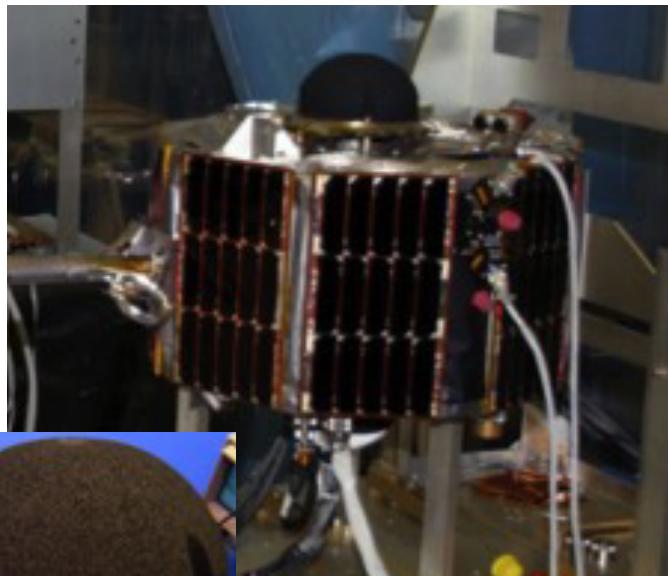
This is a result of adopting the aerodynamically excellent nose shape, reduced running resistance thanks to the drastically smoothened car body and under-floor equipment, effective

this nose ... has been newly developed ... using the latest analytical technique (i.e. **genetic algorithms**)

N700 cars save **19%** energy ... **30%** increase in the output... This is a result of adopting the ... nose shape

# Example

NASA ST5 satellite

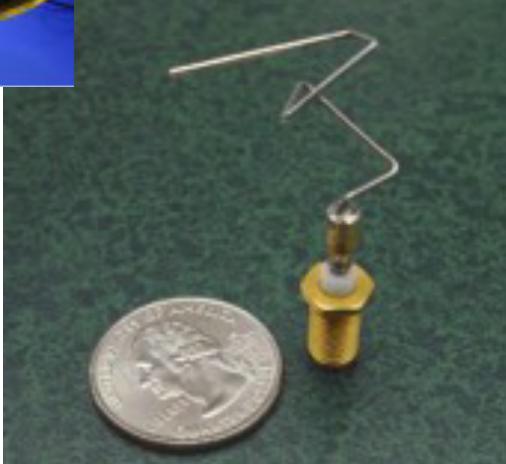
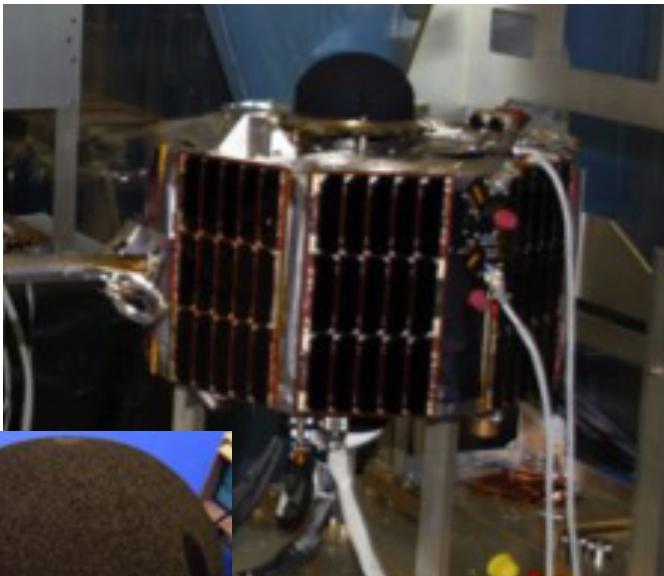


*hard to apply traditional optimization methods*



# Example

## NASA ST5 satellite



*hard to apply traditional optimization methods*

A screenshot of the Ames Research Center website. The header includes the NASA logo and links for HOME, NEWS, MISSIONS, MULTIMEDIA, CONNECT, and ABOUT NASA. The main content area displays a news article about the computer-automated evolution of an X-band antenna.

### Computer-Automated Evolution of an X-Band Antenna for NASA's Space Technology 5 Mission

**Gregory. S. Hornby**

Gregory.S.Hornby@nasa.gov  
Mail Stop 269-3, University Affiliated Research Center, UC Santa Cruz, Moffett Field, CA, 94035, USA

**Jason D. Lohn**

Jason.Lohn@west.cmu.edu  
Carnegie Mellon University, Mail Stop 23-11, Moffett Field, CA 94035, USA

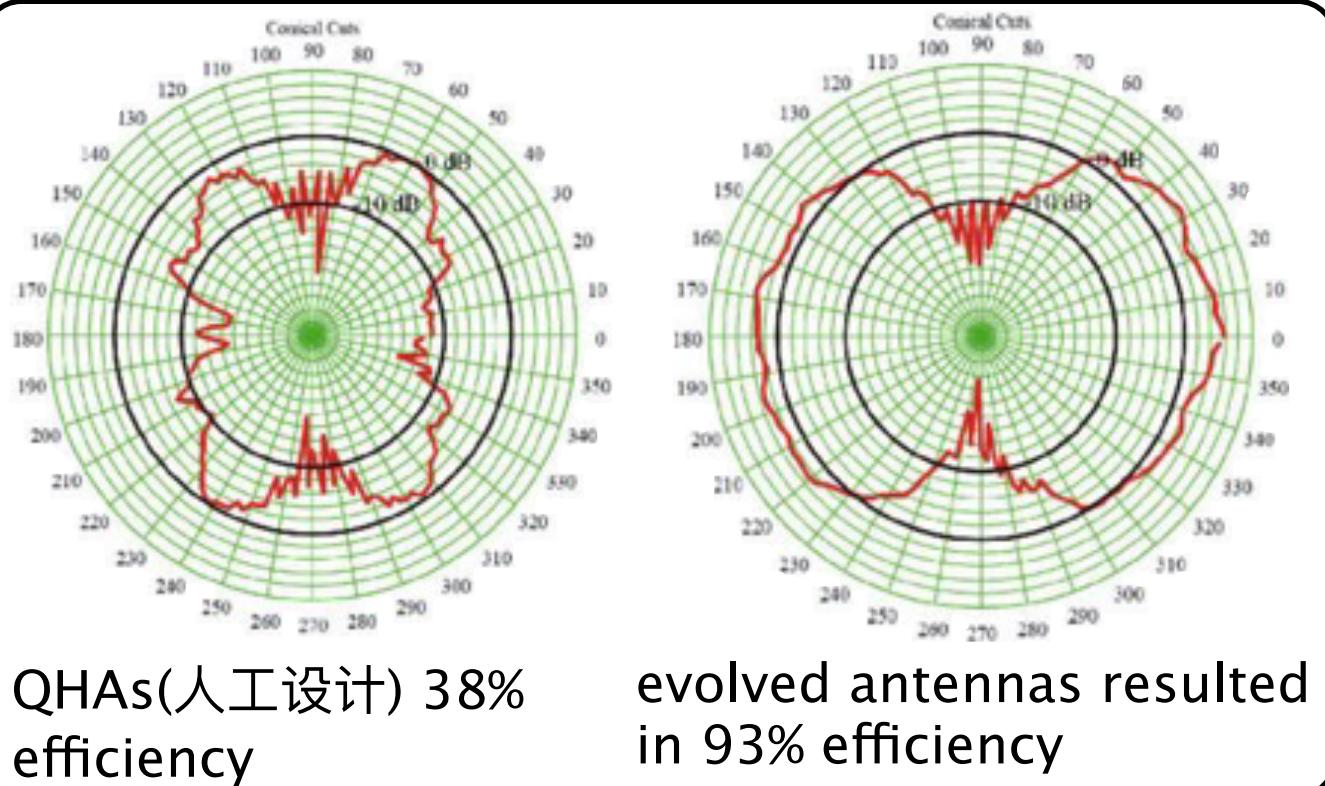
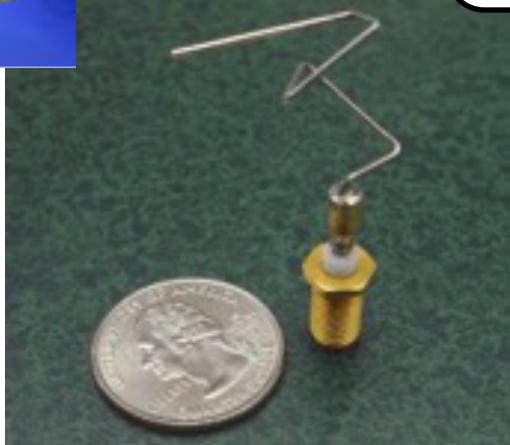
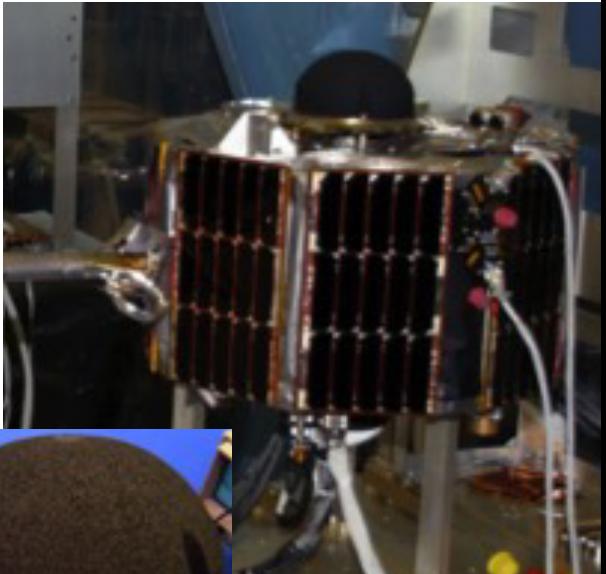
**Derek S. Linden**

dlinden@jemengineering.com  
JEM Engineering, 8683 Cherry Lane, Laurel, MD 20707, USA Moffett Field, CA 94035, USA

Since there are two antennas on each spacecraft, and not just one, it is important to measure the overall gain pattern with two antennas mounted on the spacecraft. For this, different combinations of the two evolved antennas and the QHA were tried on the ST5 mock-up and measured in an anechoic chamber. With two QHAs 38% efficiency was achieved, using a QHA with an evolved antenna resulted in 80% efficiency and using two evolved antennas resulted in 93% efficiency. Here "efficiency" means how much power is being radiated versus how much power is being eaten up in resistance, with greater efficiency resulting in a stronger signal and greater range. Figure 11

# Example

NASA ST5 satellite

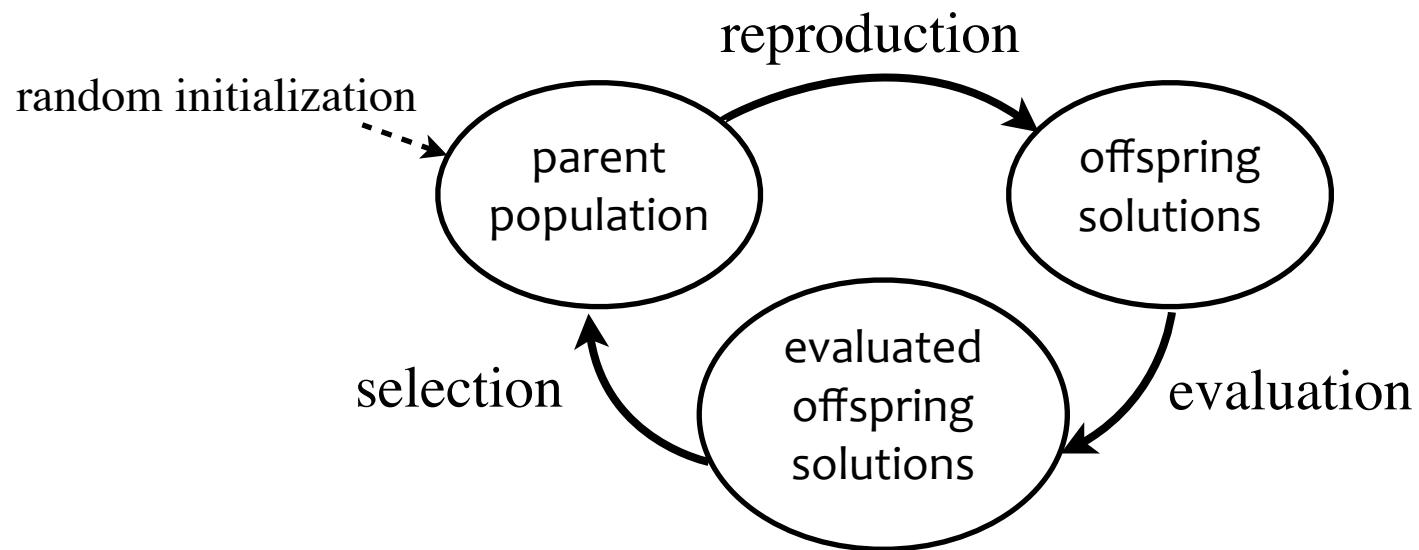


**Jason D. Lohn**  
Carnegie Mellon University, Mail Stop 23-11, Moffett Field, CA 94035, USA  
[john@west.cmu.edu](mailto:john@west.cmu.edu)

**Derek S. Linden**  
JEM Engineering, 8683 Cherry Lane, Laurel, MD 20707, USA  
[dlinden@jemengineering.com](mailto:dlinden@jemengineering.com)

Since there are two antennas on each spacecraft, and not just one, it is important to measure the overall gain pattern with two antennas mounted on the spacecraft. For this, different combinations of the two evolved antennas and the QHA were tried on the ST5 mock-up and measured in an anechoic chamber. With two QHAs 38% efficiency was achieved, using a QHA with an evolved antenna resulted in 80% efficiency and using two evolved antennas resulted in 93% efficiency. Here "efficiency" means how much power is being radiated versus how much power is being eaten up in resistance, with greater efficiency resulting in a stronger signal and greater range. Figure 11

# Properties of meta-heuristics



zeroth order

do not need differentiable functions

convergence

will find an optimal solution if  
or  $P( x \rightarrow x_1 \rightarrow \dots \rightarrow x_k \rightarrow x^* ) > 0$

$$P( x^* \mid x ) > 0$$

# Which is the best algorithm?

a search algorithm  $A$ , objective  $f$ ,  $m$  solutions

arbitrary measure of the objective values of the  $m$  solutions:

$$\Phi(\mathbf{y}_m \mid f, m, A)$$

assume no replicates

Over all objectives     $f : \mathcal{X} \rightarrow \{1, 2, \dots, Y\}$

Overall performance assessment (for arbitrary  $k$ ):

$$\sum_f I[k = \Phi(\mathbf{y}_m \mid f, m, A)]$$

$$\sum_f I[k = \Phi(\mathbf{y}_m | f, m, A)] = \sum_f I[k = \Phi(f(A(m)))]$$

$$\begin{aligned} \sum_f I[k = \Phi(\mathbf{y}_m | f, m, A)] &= \sum_f I[k = \Phi(f(A(m)))] \\ &= \sum_f \sum_{\mathbf{y}_m} I[k = \Phi(\mathbf{y}_m)] I[\mathbf{y}_m = f(A(m))] \end{aligned}$$

$$\begin{aligned} \sum_f I[k = \Phi(\mathbf{y}_m | f, m, A)] &= \sum_f I[k = \Phi(f(A(m)))] \\ &= \sum_f \sum_{\mathbf{y}_m} I[k = \Phi(\mathbf{y}_m)] I[\mathbf{y}_m = f(A(m))] \\ &= \sum_{\mathbf{y}_m} I[k = \Phi(\mathbf{y}_m)] \sum_f I[\mathbf{y}_m = f(A(m))] \end{aligned}$$

$$\begin{aligned} & \sum_f I[k = \Phi(\mathbf{y}_m | f, m, A)] = \sum_f I[k = \Phi(f(A(m)))] \\ &= \sum_f \sum_{\mathbf{y}_m} I[k = \Phi(\mathbf{y}_m)] I[\mathbf{y}_m = f(A(m))] \\ &= \sum_{\mathbf{y}_m} I[k = \Phi(\mathbf{y}_m)] \sum_f I[\mathbf{y}_m = f(A(m))] \\ &= \sum_{\mathbf{y}_m} I[k = \Phi(\mathbf{y}_m)] Y^{|\mathcal{X}| - m} \end{aligned}$$

$$\begin{aligned} & \sum_f I[k = \Phi(\mathbf{y}_m | f, m, A)] = \sum_f I[k = \Phi(f(A(m)))] \\ &= \sum_f \sum_{\mathbf{y}_m} I[k = \Phi(\mathbf{y}_m)] I[\mathbf{y}_m = f(A(m))] \\ &= \sum_{\mathbf{y}_m} I[k = \Phi(\mathbf{y}_m)] \sum_f I[\mathbf{y}_m = f(A(m))] \\ &= \sum_{\mathbf{y}_m} I[k = \Phi(\mathbf{y}_m)] Y^{|\mathcal{X}| - m} \end{aligned}$$

all algorithms have the same average performance

[Wolpert & Macready, 97]

# No Free Lunch Theorem

$$\begin{aligned} \sum_f I[k = \Phi(\mathbf{y}_m | f, m, A)] &= \sum_f I[k = \Phi(f(A(m)))] \\ &= \sum_f \sum_{\mathbf{y}_m} I[k = \Phi(\mathbf{y}_m)] I[\mathbf{y}_m = f(A(m))] \\ &= \sum_{\mathbf{y}_m} I[k = \Phi(\mathbf{y}_m)] \sum_f I[\mathbf{y}_m = f(A(m))] \\ &= \sum_{\mathbf{y}_m} I[k = \Phi(\mathbf{y}_m)] Y^{|\mathcal{X}| - m} \end{aligned}$$

all algorithms have the same average performance

[Wolpert & Macready, 97]