## Iterative improvement

- In many optimization problems, **PATH** is irrelevant; the goal state itself is the solution.
- Then, state space = space of "complete" configurations. Algorithm goal:

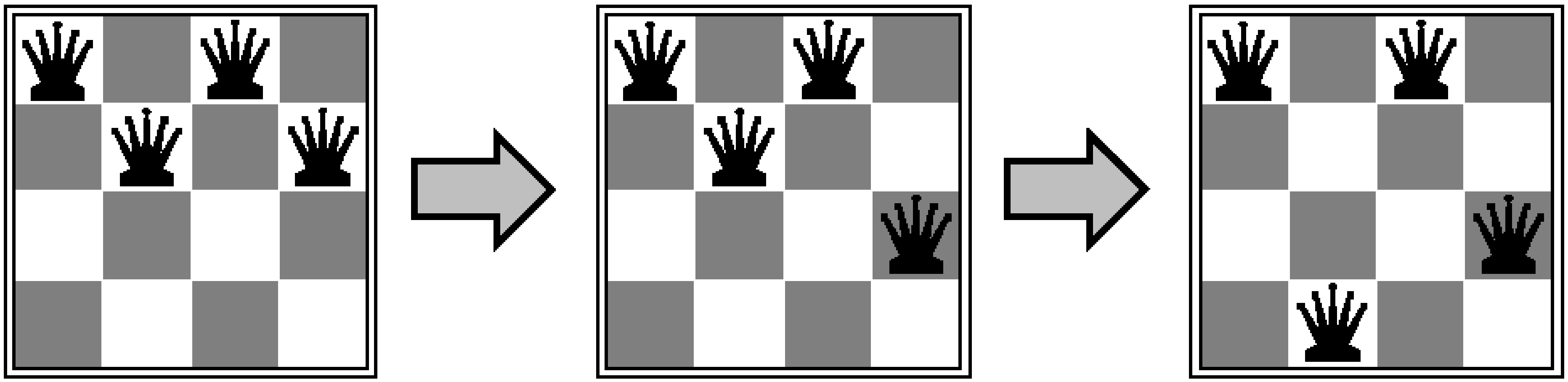
  - find configuration satisfying constraints (e.g., n-queens)
- In such cases, can use iterative improvement algorithms: keep a single "**current**" state, and try to improve it.

  - find optimal configuration (e.g., TSP ➔ find *any* solution and then improve it incrementally)

  - advantage: does better with limited time than other algorithms

## Iterative improvement example: n-queens

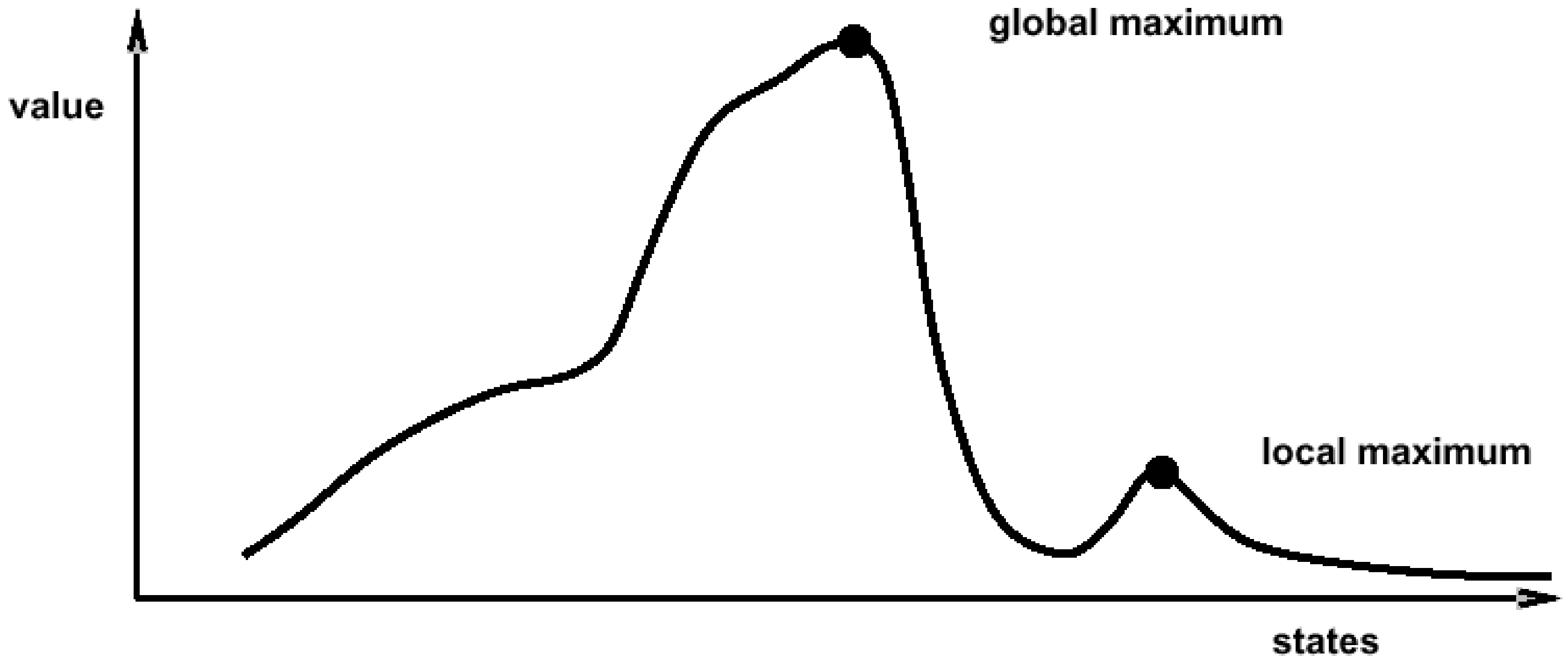
- Goal: Put n chess-game queens on an n x n board, with no two queens on the same row, column, or diagonal.



- Here, goal state is initially unknown but is specified by constraints that it must satisfy.

## Objective Functions: Local/Global Extrema

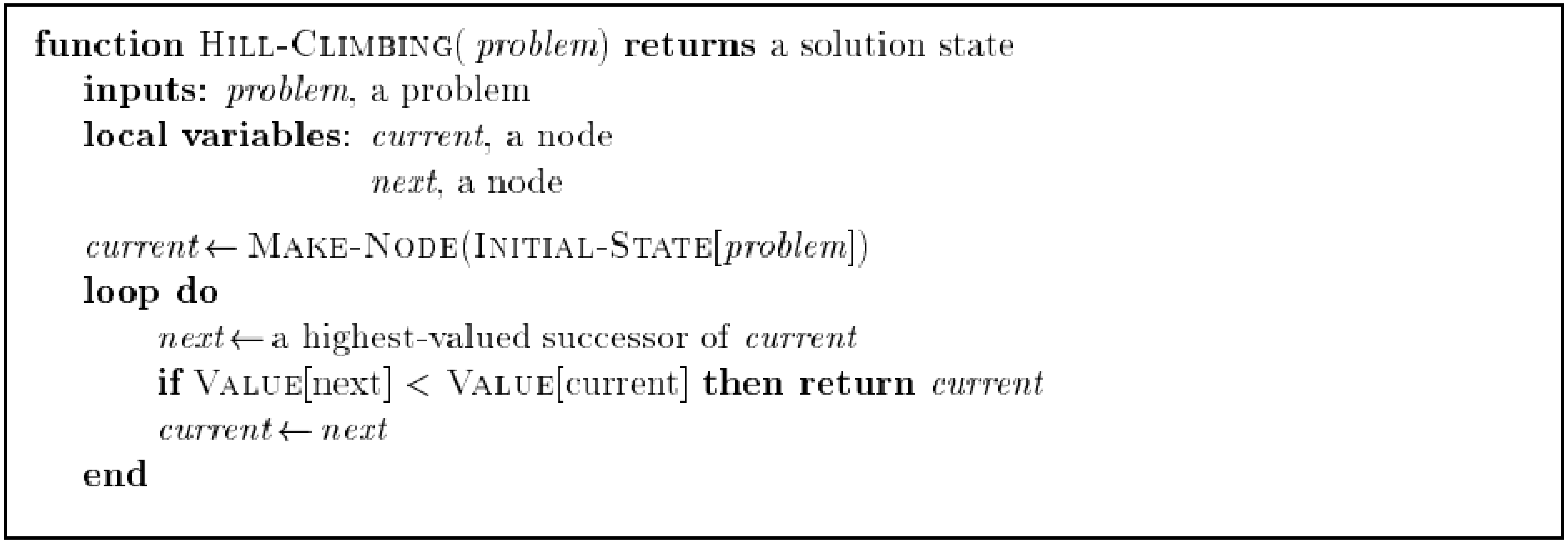
- Many optimization problems have this kind of landscape
- Local Search: finds a local max/min
- Optimal Search: finds global max/min



## Local Search: Hill climbing (or gradient ascent/descent)

- Iteratively maximize “value” of current state, by replacing it by successor state that has highest value, as long as possible. ➔ It is a “greedy” LOCAL search algorithm

“Like climbing Everest in thick fog with amnesia”



```
function HILL-CLIMBING( problem) returns a solution state
    inputs: problem, a problem
    local variables: current, a node
                     next, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        next ← a highest-valued successor of current
        if VALUE[next] < VALUE[current] then return current
        current ← next
    end
```

## Hill climbing

- Note: minimizing a “value” function v(n) is equivalent to maximizing –v(n),

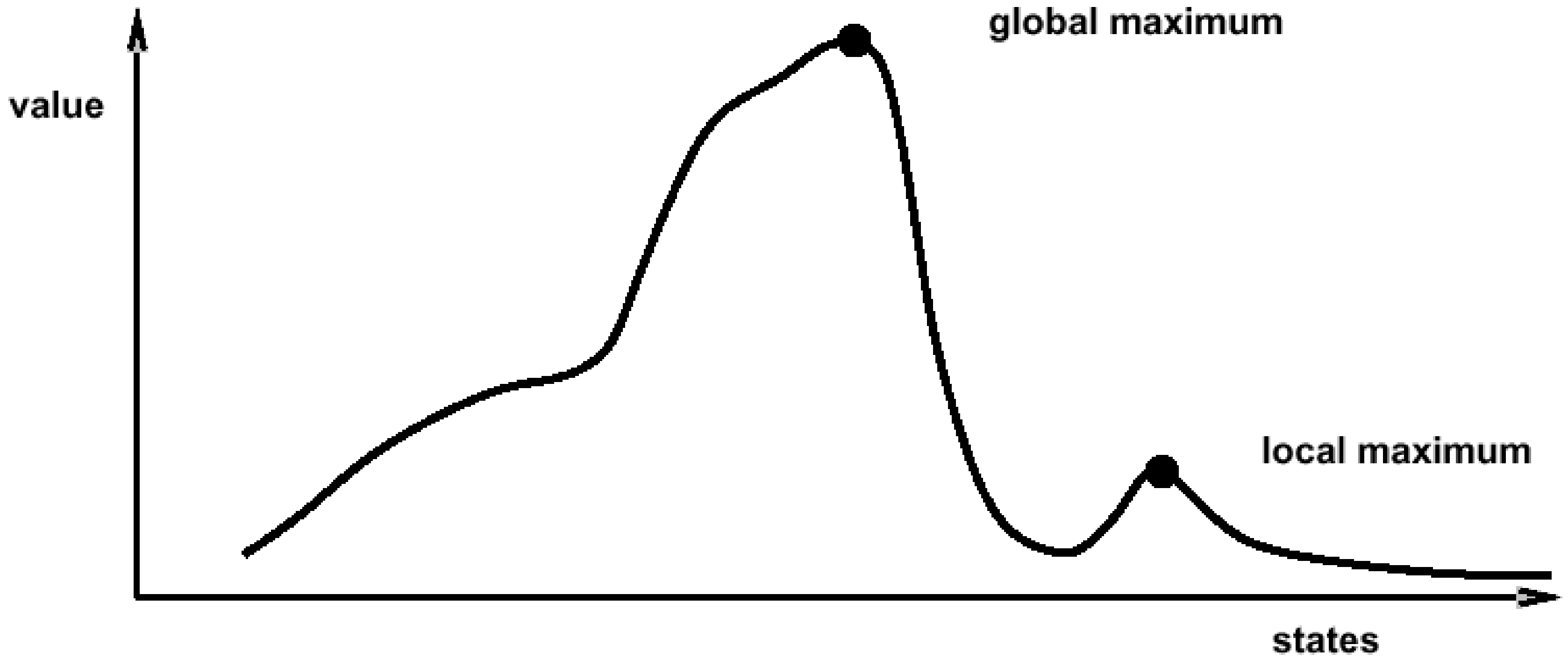
  thus both notions are used interchangeably.

- Basically greedy, local search.
- No need for a search tree.  Just go from current state to “best” successor based on heuristic function.

- Notion of “extremization”: find extrema (minima or maxima) of a value function.

## Problem with Hill climbing

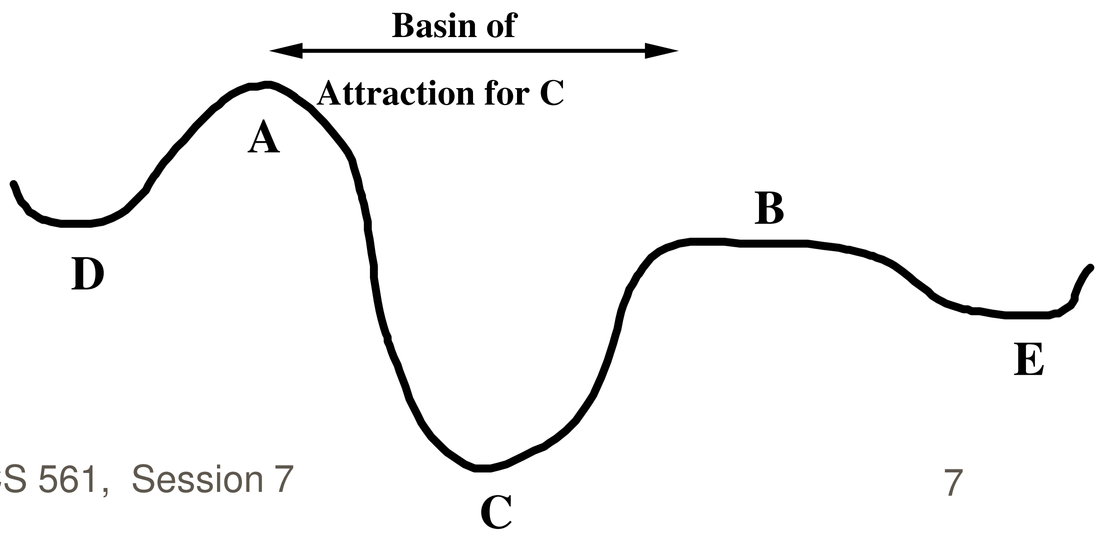
- Problem: depending on initial state, may get stuck in the local extremum.



## Minimizing energy

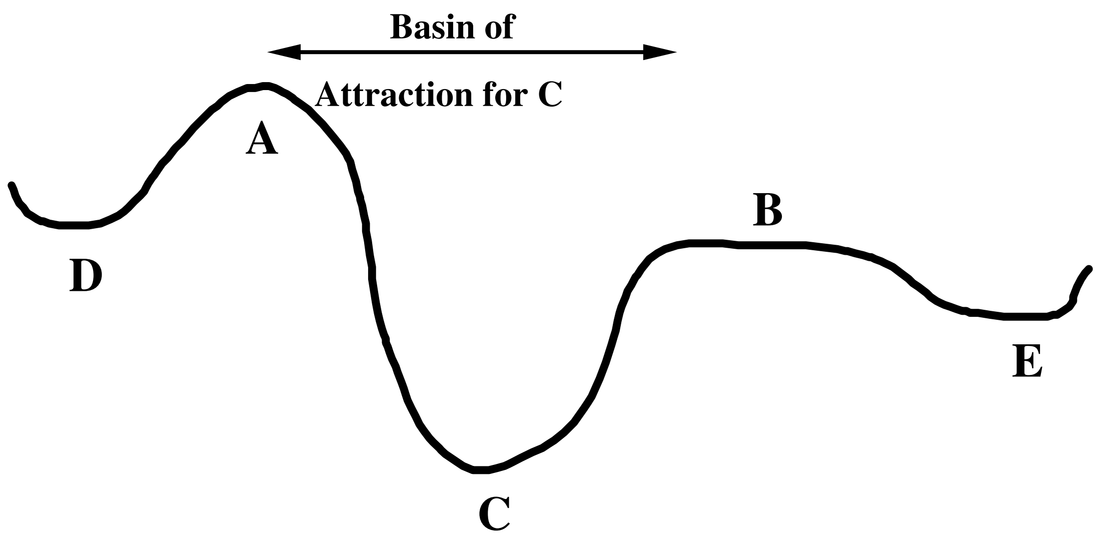
- Let's now change the formulation of the problem a bit, so that we can employ new formalism:
  - let us compare our state space to that of a physical system that is subject to natural interactions,
  - and let us compare our value function to the overall potential energy E of the system.

- On every updating, we have $\Delta E \leq 0$



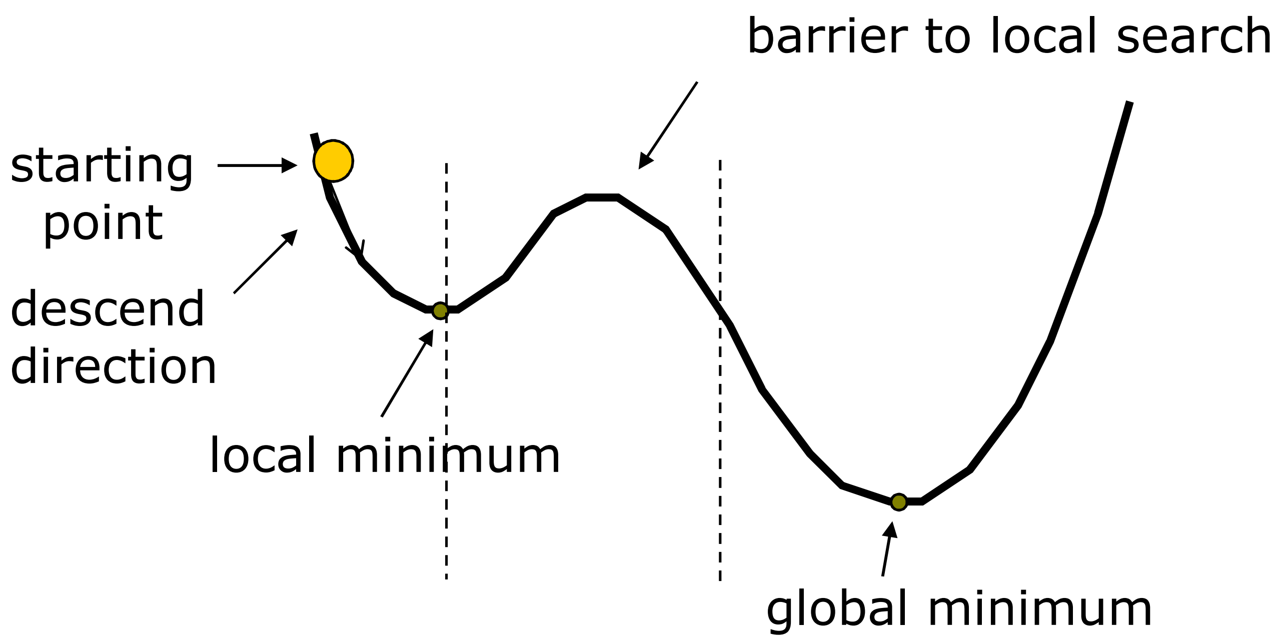
## Minimizing energy

- Hence the dynamics of the system tend to move E toward a minimum.
- We stress that there may be different such states — they are *local* minima. Global minimization is not guaranteed.



## Local Minima Problem

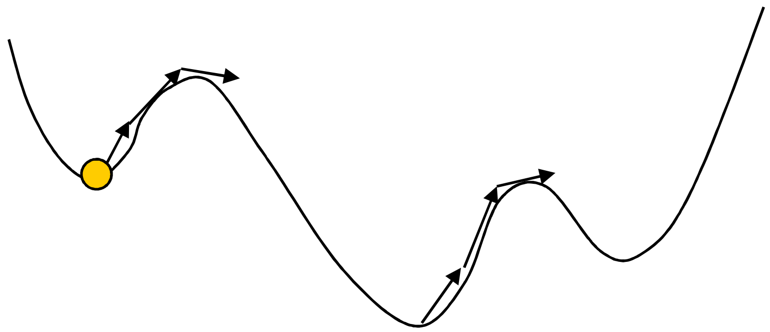
- Question: How do you avoid this local minimum?



## Consequences of the Occasional Ascents

desired effect

Help escaping the local optima.



adverse effect

Might pass global optima after reaching it

(easy to avoid by keeping track of best-ever state)

## Boltzmann’s statistical theory of gases

- In the statistical theory of gases, the gas is described not by a deterministic dynamics, but rather by the probability that it will be in different states.

- The 19th century physicist Ludwig Boltzmann developed a theory that included a probability distribution of temperature (i.e., every small region of the gas had the same kinetic energy).

- Hinton, Sejnowski and Ackley’s idea was that this distribution might also be used to describe neural interactions, where low temperature T is replaced by a small noise term T (the neural analog of random thermal motion of molecules). While their results primarily concern optimization using neural networks, the idea is more general.

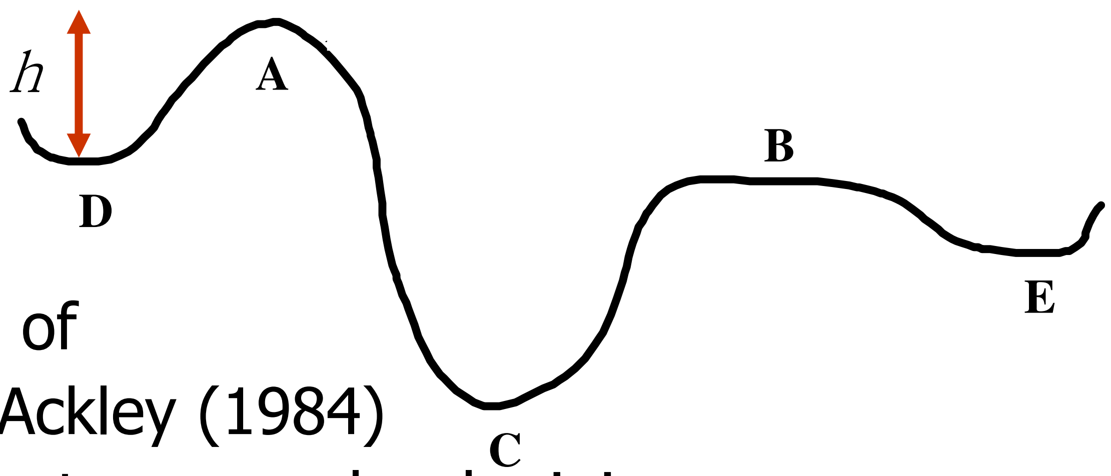
## Boltzmann distribution

- At thermal equilibrium at temperature T, the Boltzmann distribution gives the relative probability that the system will occupy state A vs. state B as:

$$\frac{P(A)}{P(B)} = \exp\left(-\frac{E(A)-E(B)}{T}\right) = \frac{\exp(E(B)/T)}{\exp(E(A)/T)}$$

- where E(A) and E(B) are the energies associated with states A and B.

## Boltzmann machines



The Boltzmann Machine of
Hinton, Sejnowski, and Ackley (1984)
uses simulated annealing to escape local minima.

To motivate their solution, consider how one might get a ball-bearing traveling along the curve to "probably end up" in the deepest minimum. The idea is to shake the box "about h hard" — then the ball is more likely to go from D to C than from C to D. So, on average, the ball should end up in C's valley.

## Simulated annealing

Kirkpatrick et al. 1983:

- Simulated annealing is a general method for making likely the escape from local minima by allowing jumps to higher energy states.
- The analogy here is with the process of annealing used by a craftsman in forging a sword from an alloy.
- He heats the metal, then slowly cools it as he hammers the blade into shape.
  - If he cools the blade too quickly the metal will form patches of different composition;
  - If the metal is cooled slowly while it is shaped, the constituent metals will form a uniform alloy.

## Simulated annealing: basic idea

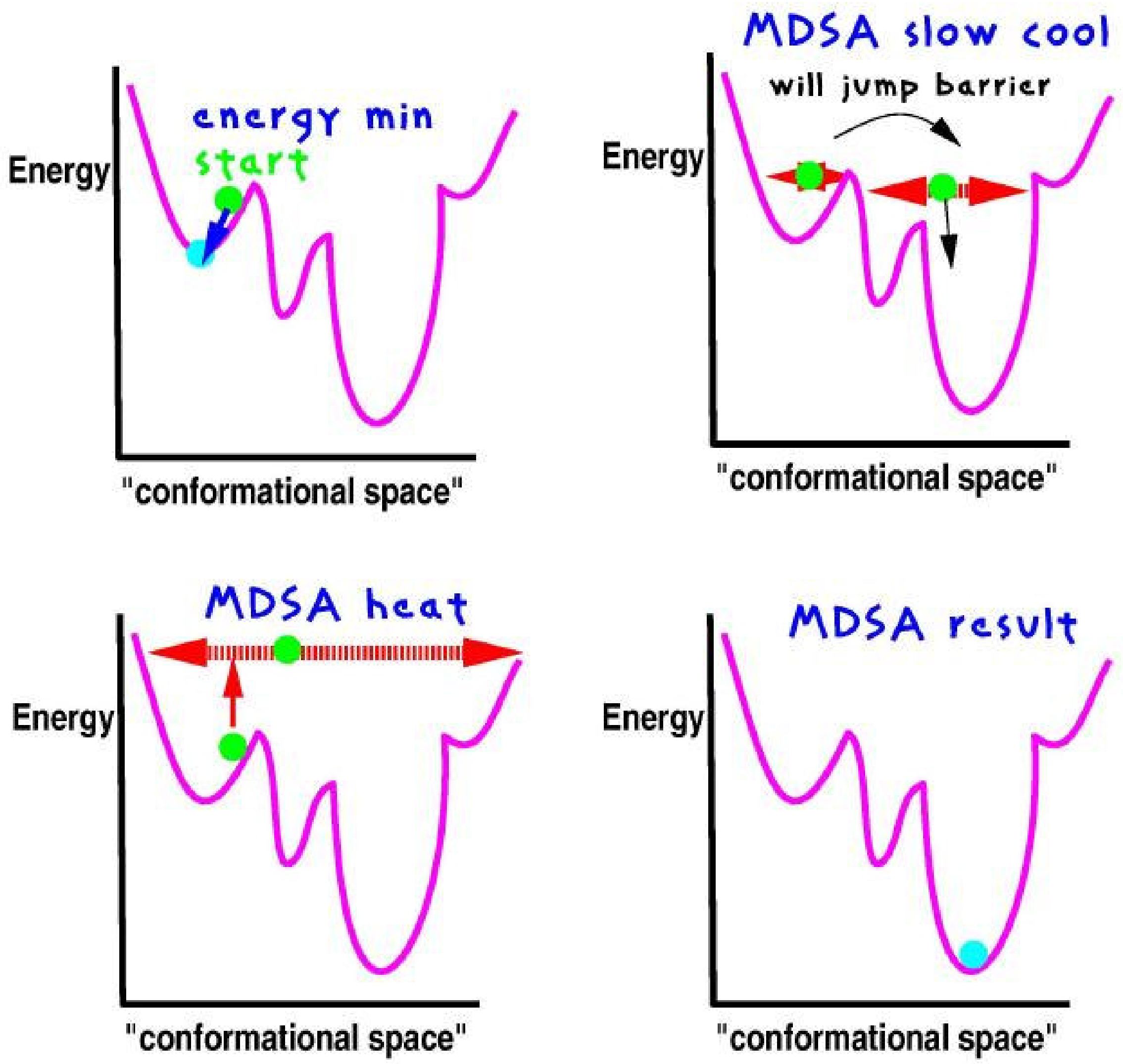
- From current state, pick a **random** successor state;
- If it has better value than current state, then "accept the transition," that is, use successor state as current state;
- Otherwise, do not give up, but instead flip a coin and accept the transition with a given probability (that is lower as the successor is worse).
- So we accept to sometimes "un-optimize" the value function a little with a non-zero probability.

## Simulated annealing in practice

- set T
- optimize for given T
- lower T (see Geman & Geman, 1984)
- repeat

## Simulated annealing in practice

- set T
- optimize for given T
- lower T
- repeat



MDSA: Molecular Dynamics Simulated Annealing

## Simulated annealing in practice

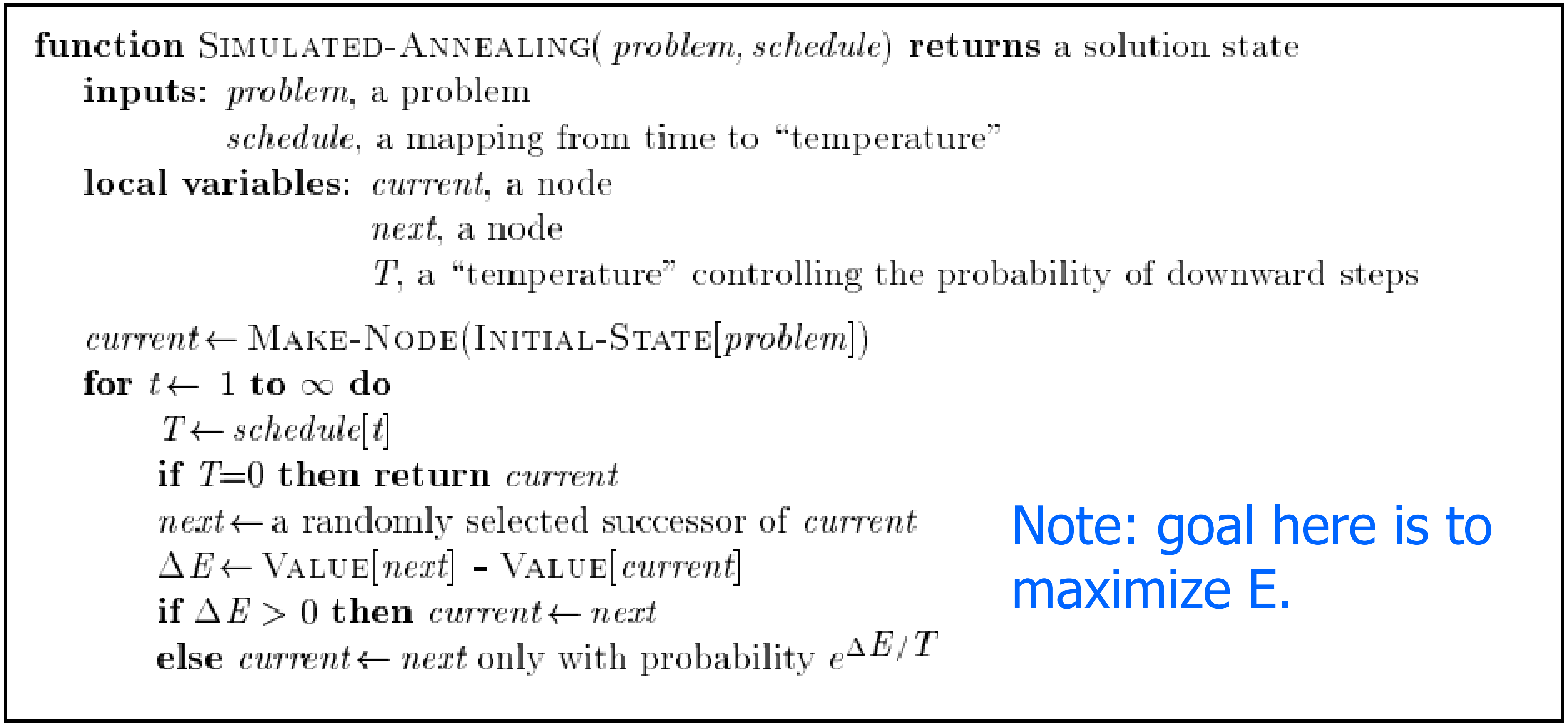
- set T
- optimize for given T
- lower T (see Geman & Geman, 1984)
- repeat

- Geman & Geman (1984): if T is lowered sufficiently slowly (with respect to the number of iterations used to optimize at a given T), simulated annealing is guaranteed to find the global minimum.

- Caveat: this algorithm has no end (Geman & Geman's T decrease schedule is in the 1/log of the number of iterations, so, T will never reach zero), so it may take an infinite amount of time for it to find the global minimum.

## Simulated annealing algorithm

- Idea: Escape local extrema by allowing “bad moves,” but gradually decrease their size and frequency.



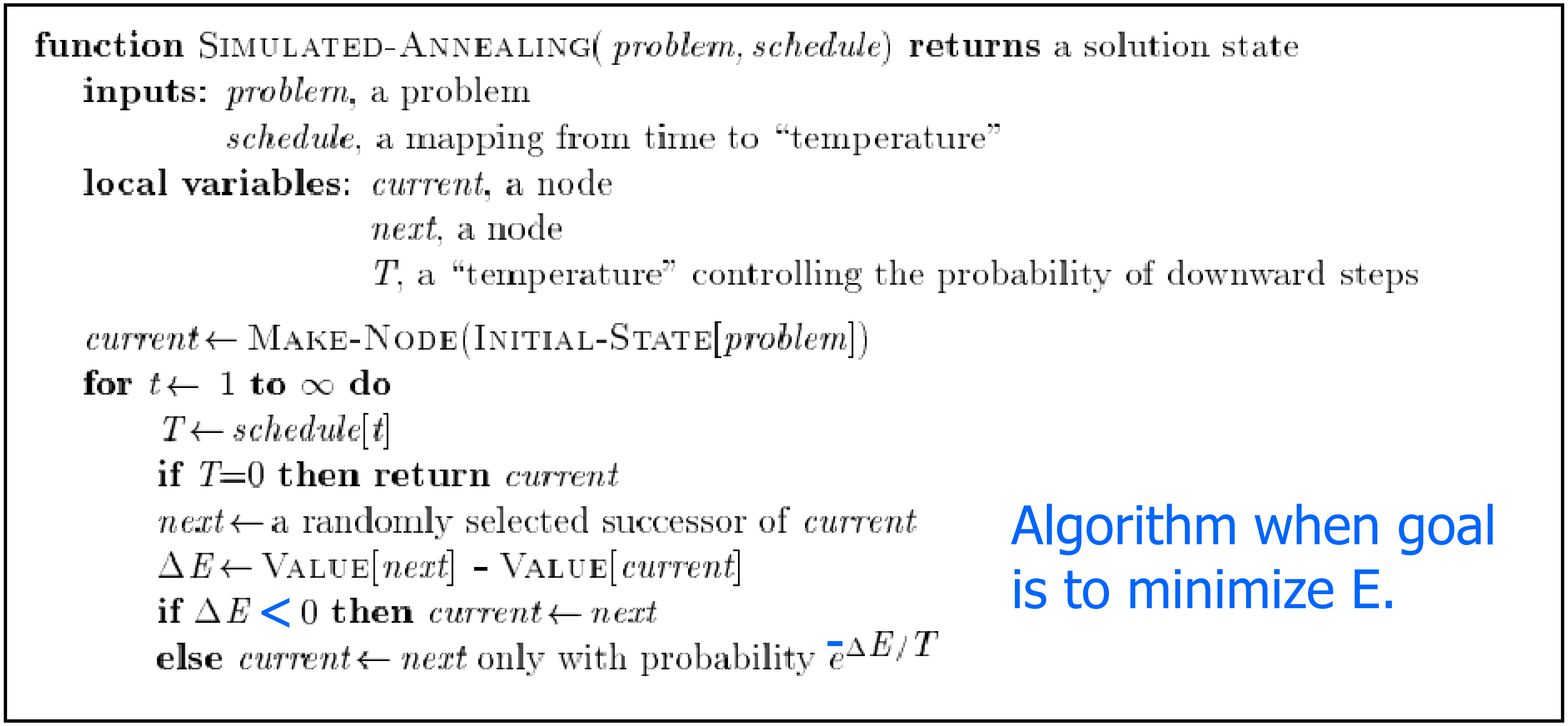
```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to “temperature”
   local variables: current, a node
                    next, a node
                    T, a “temperature” controlling the probability of downward steps

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

Note: goal here is to maximize E.

## Simulated annealing algorithm

- Idea: Escape local extrema by allowing “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 the probability of downward steps

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

Algorithm when goal is to minimize E.

## Note on simulated annealing: limit cases

- Boltzmann distribution: accept “bad move” with $\Delta E<0$ (goal is to maximize E) with probability $P(\Delta E) = \exp(\Delta E/T)$

- If T is large:
  $\Delta E < 0$
  $\Delta E/T < 0$ and very small
  $\exp(\Delta E/T)$ close to 1
  accept bad move with high probability

- If T is near 0:
  $\Delta E < 0$
  $\Delta E/T < 0$ and very large
  $\exp(\Delta E/T)$ close to 0
  accept bad move with low probability

## Note on simulated annealing: limit cases

- Boltzmann distribution: accept "bad move" with $\Delta E<0$ (goal is to maximize E) with probability $P(\Delta E) = \exp(\Delta E/T)$

- If T is large:
  - $\Delta E < 0$
  - $\Delta E/T < 0$ and very small
  - $\exp(\Delta E/T)$ close to 1
  - accept bad move with high probability

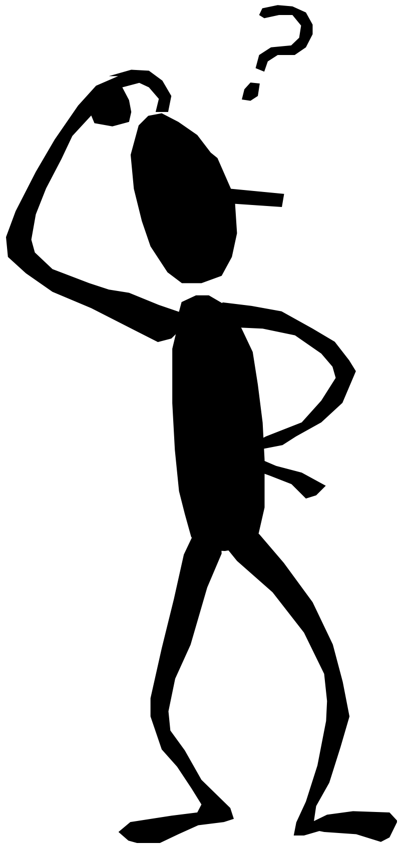
**Random walk**

- If T is near 0:
  - $\Delta E < 0$
  - $\Delta E/T < 0$ and very large
  - $\exp(\Delta E/T)$ close to 0
  - accept bad move with low probability

**Deterministic down-hill**

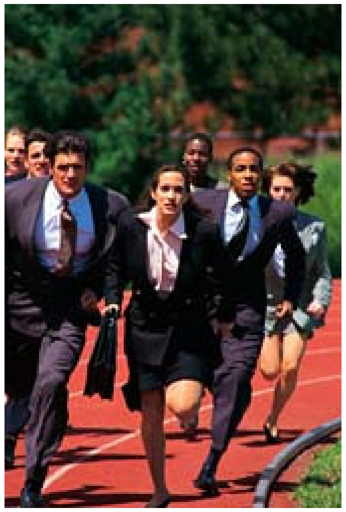
## How do you find a solution in a large complex space?

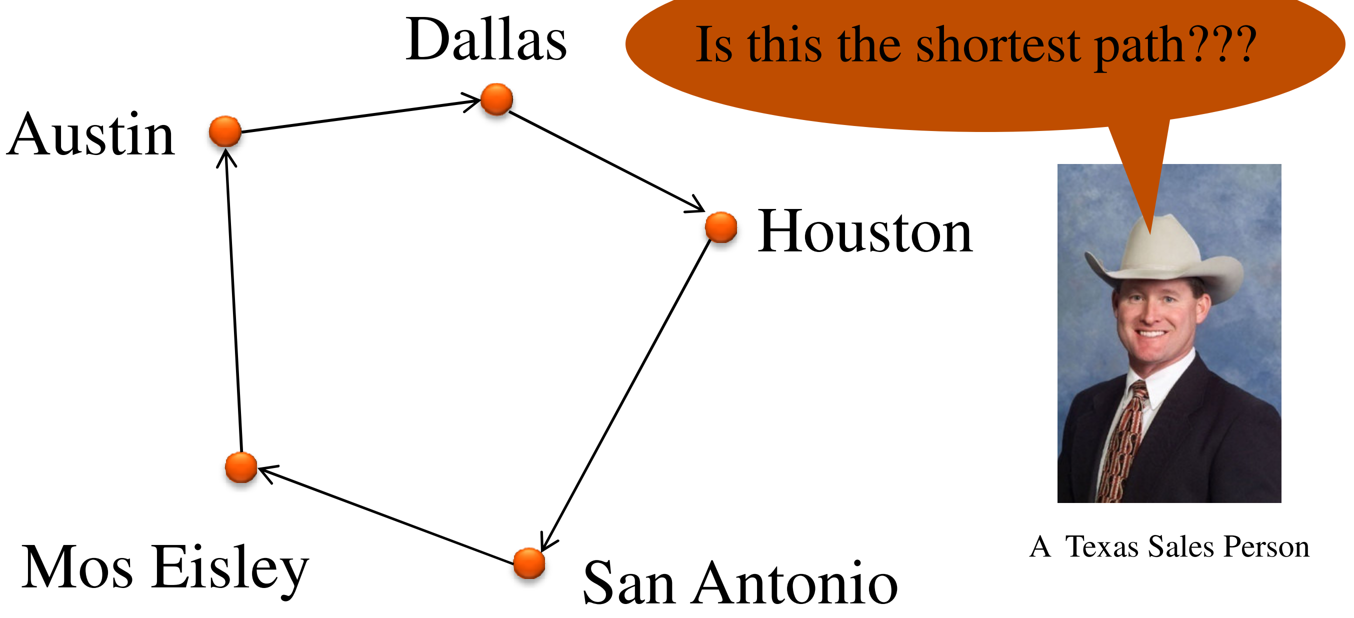
- Ask an expert?
- Adapt existing designs?
- Trial and error?



## Example: Traveling Sales Person (TSP)

- Classic Example: You have N cities, find the shortest route such that your salesperson will visit each city once and return.
- This problem is known to be NP-Hard
  - As a new city is added to the problem, computation time in the classic solution increases exponentially *$O(2^n)$ … (as far as we know)*





A Texas Sales Person

## What if………

- Lets create a whole bunch of random sales people and see how well they do and pick the best one(s).
  - Salesperson A
    - Houston -> Dallas -> Austin -> San Antonio -> Mos Eisely
    - Distance Traveled 780 Km
  - Salesperson B
    - Houston -> Mos Eisley -> Austin -> San Antonio -> Dallas
    - Distance Traveled 820 Km
  - Salesperson A is better (more fit) than salesperson B
  - Perhaps we would like sales people to be more like **A** and less like **B**
- Question:
  - do we want to just keep picking random sales people like this and keep testing them?

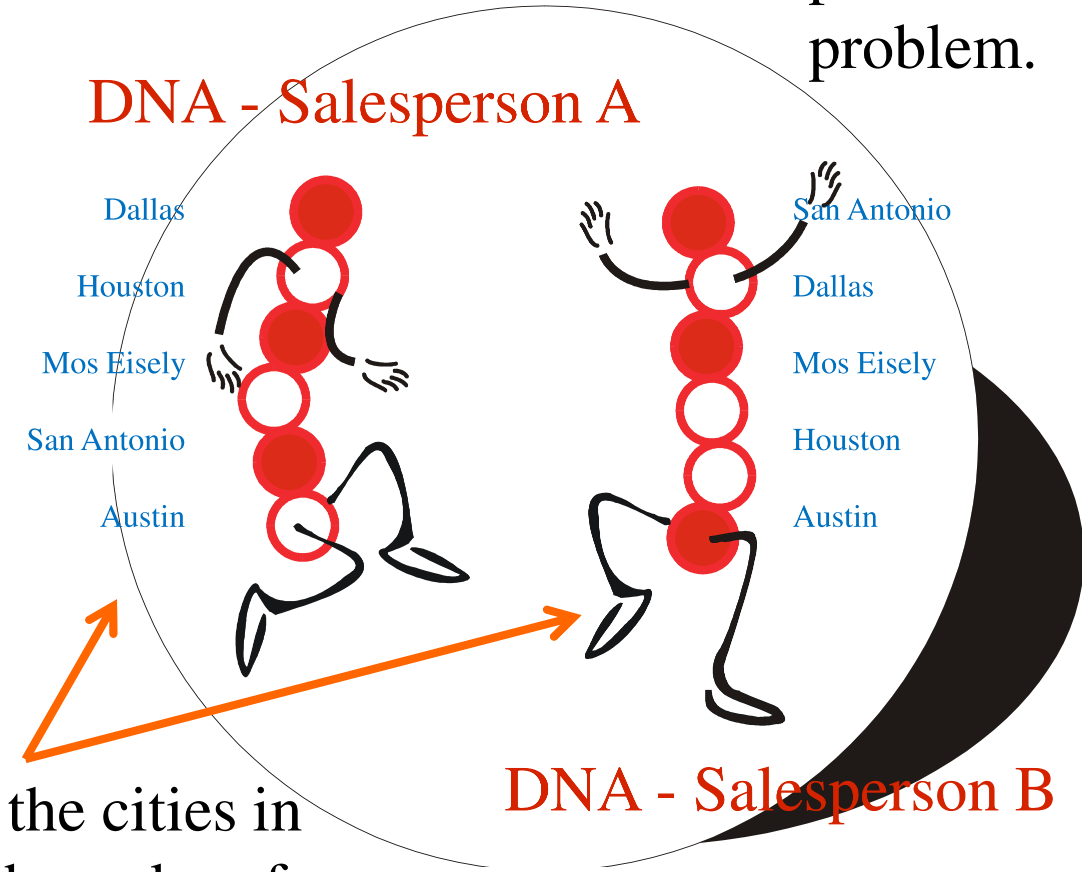
## We can get a little closer to the solution in polynomial time

- We might use a heuristic(s) to guide us in creating new sales people
  - So for instance, we might use the triangle inequality to help pick better potential sales people.
  - One can create an initial 2 approximation (at worst the distance is twice the optimal distance) to TSP using a Nearest Neighbor or other similar efficient polynomial time method.
  - This detail is somewhat unimportant, you can use different heuristics to help you create a better initial set of sales people

- Use some sort of incremental improvement to make them better successively.
  - The idea is that you start with result(s) closer to where you think the solution is than one would obtain at random so that the problem **converges more quickly**.
  - **Be careful** since an initial approximation may be too close to a local extrema which might actually slow down convergence or throw the solution off.

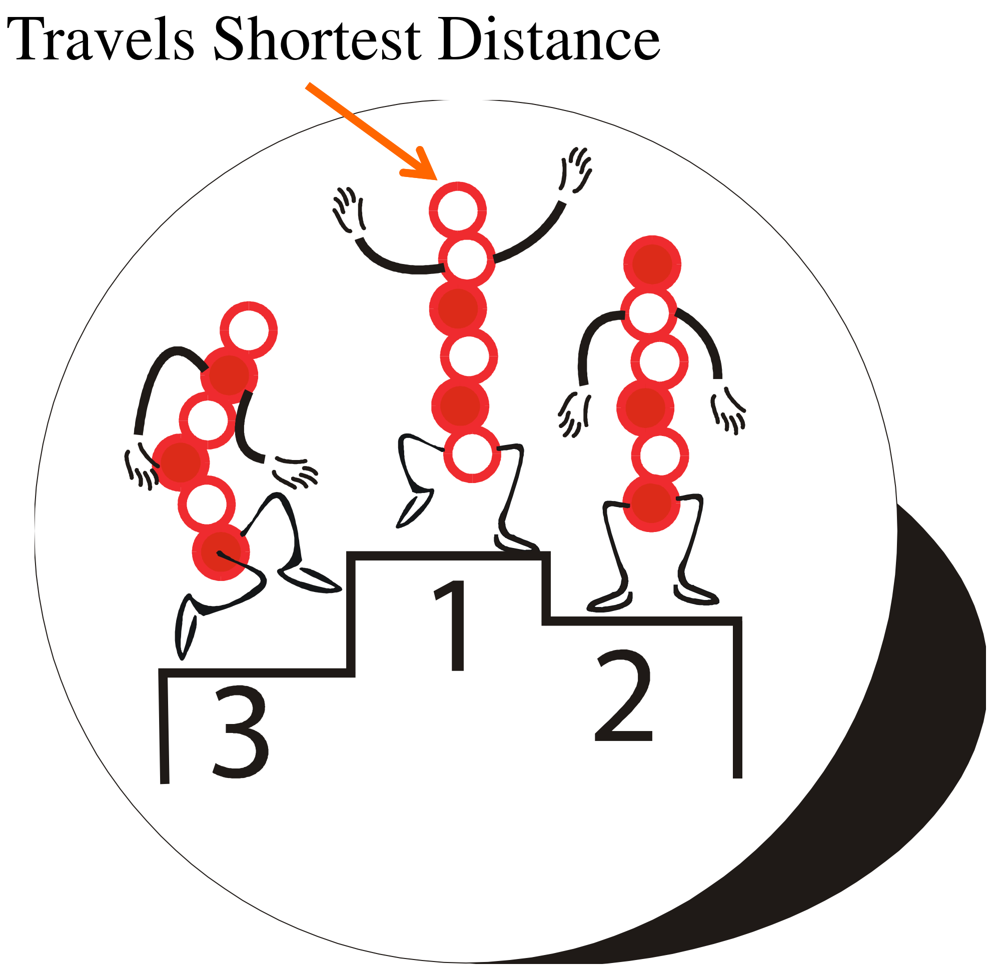
## Represent problem like a DNA sequence

Each DNA sequence is a possible solution to the problem.



The order of the cities in the genes is the order of the cities the TSP will take.

## Ranking by Fitness:



Here we've created three different salespeople. We then checked to see how far each one has to travel. This gives us a measure of "Fitness"

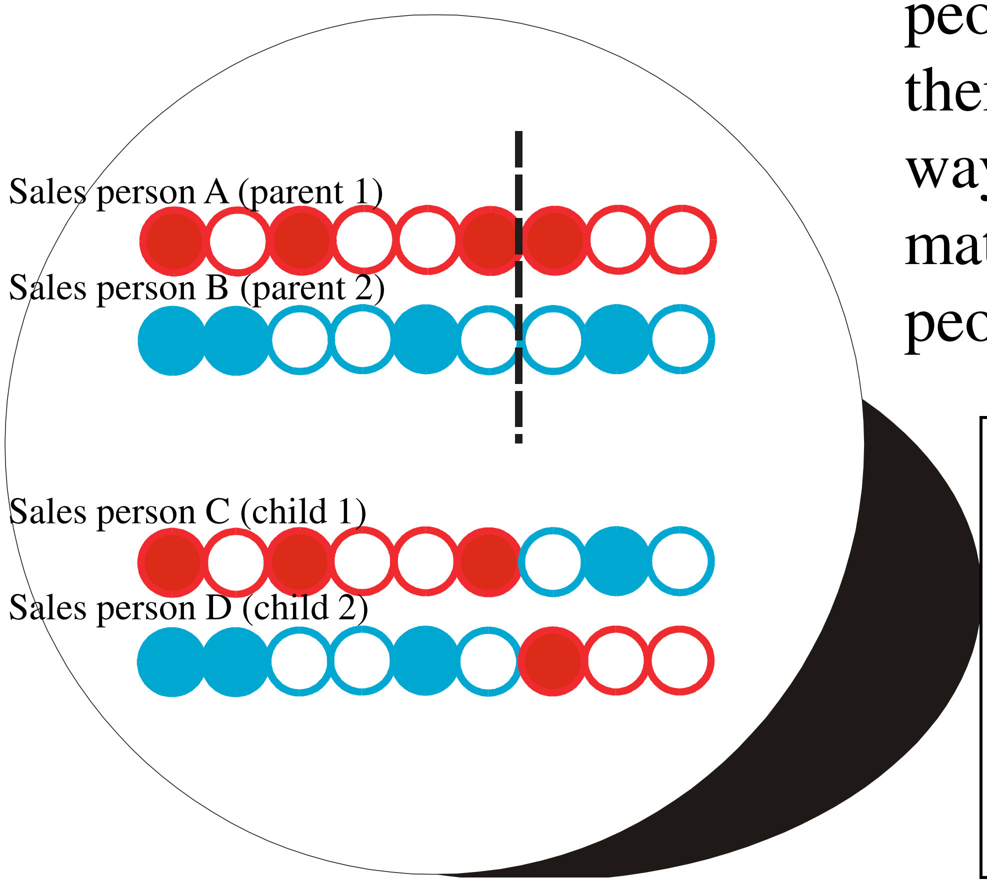
Note: we need to be able to measure fitness in polynomial time, otherwise we are in trouble.

## Let’s breed them!

- We have a population of traveling sales people. We also know their fitness based on how long their trip is. We want to create more, but we don’t want to create too many.
- We take the notion that the salespeople who perform better are closer to the optimal salesperson than the ones which performed more poorly. Could the optimal sales person be a “combination” of the better sales people?
- We create a population of sales people as solutions to the problem.
- *How* do we actually mate a population of data???

**Crossover:**

Exchanging information through some part of information (representation)

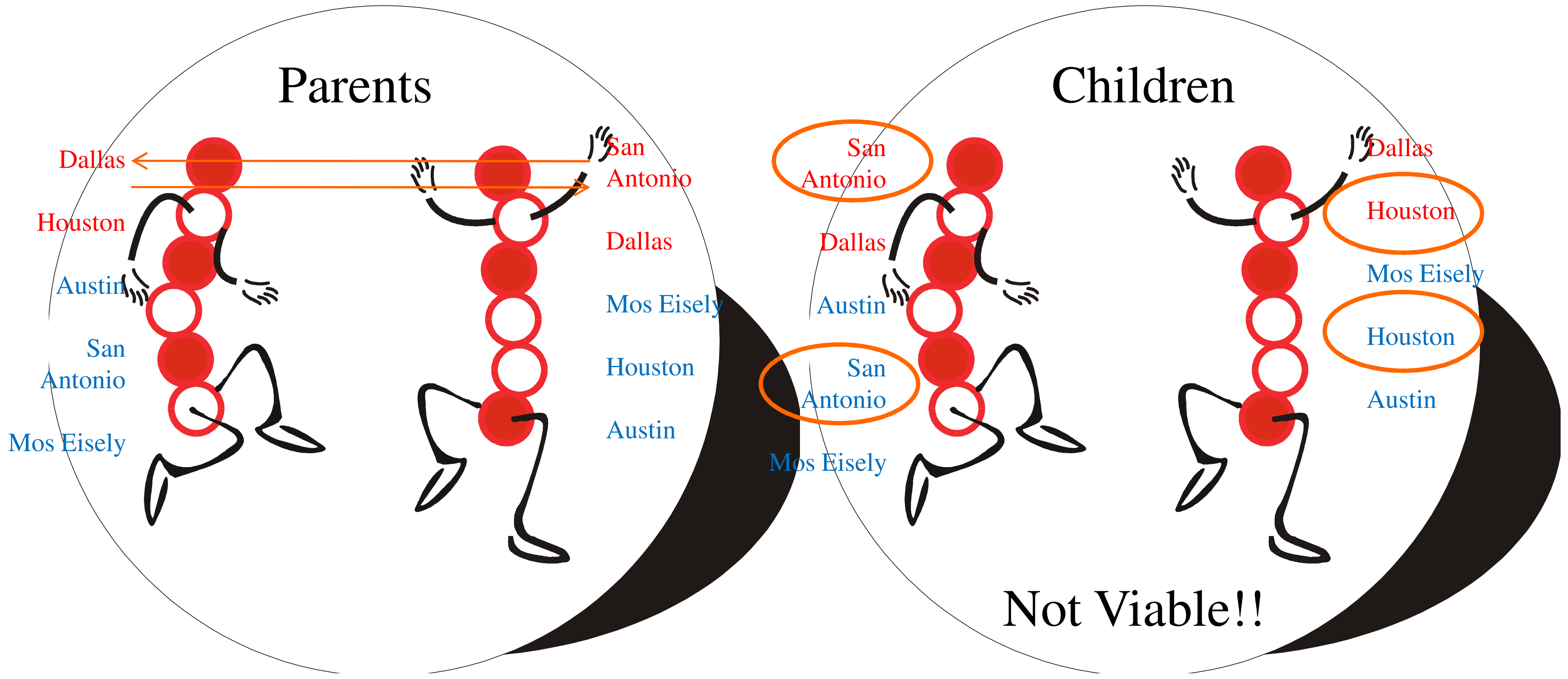


Once we have found the best sales people we will in a sense mate them. We can do this in several ways. Better sales people should mate more often and poor sales people should mate lest often.

| Sales People | City DNA |
|---|---|
| Parent 1 | F A B \| E C G D |
| Parent 2 | D E A \| C G B F |
| Child 1 | F A B \| C G B F |
| Child 2 | D E A \| E C G D |

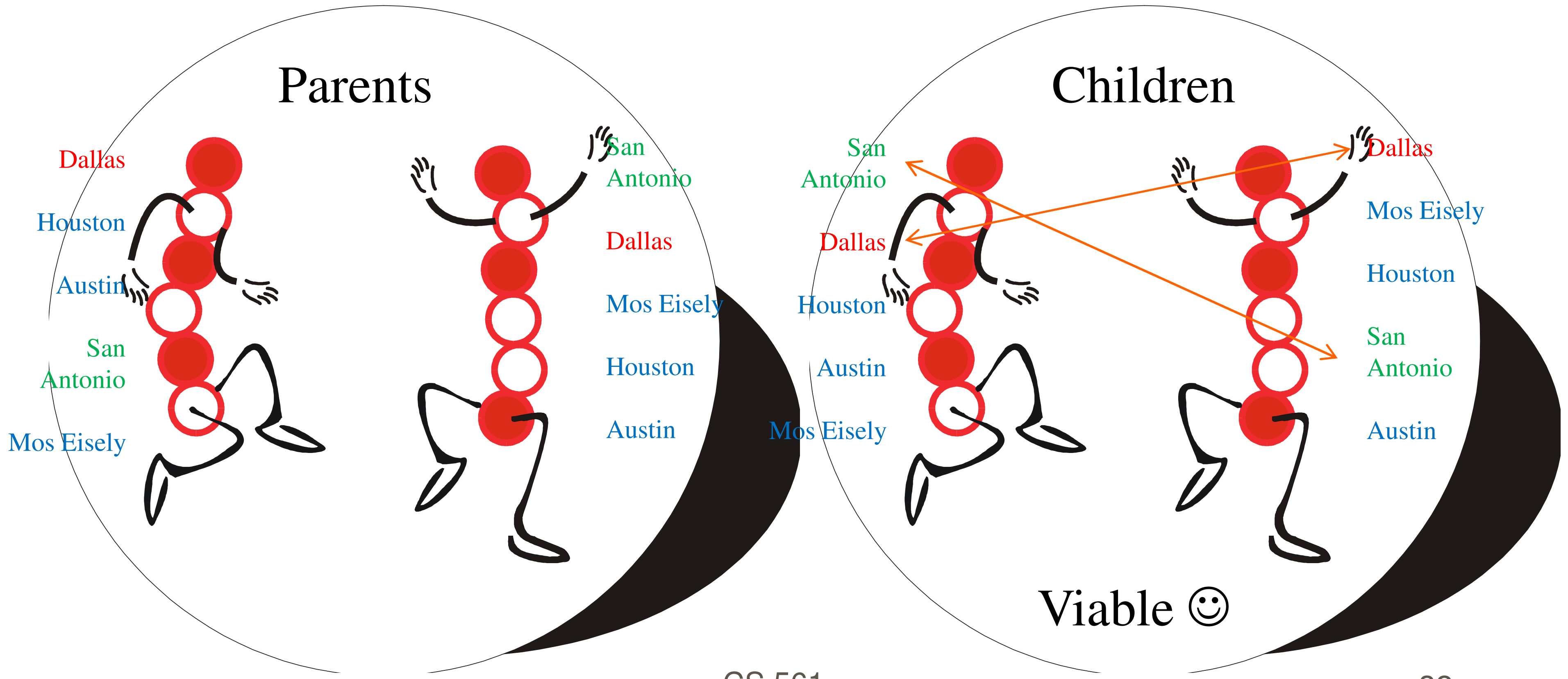
## Crossover Bounds (Houston we have a problem)

- Not all crossed pairs are viable. We can only visit a city once.
- Different GA problems may have different bounds.



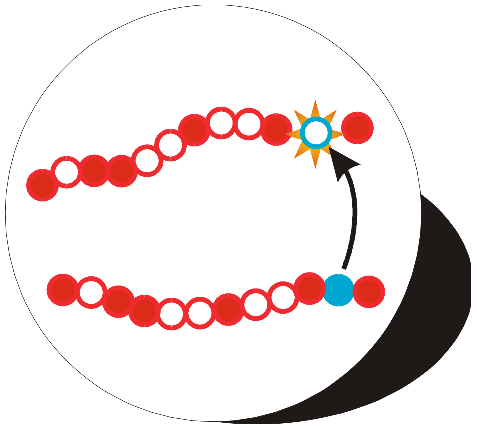
## TSP needs some special rules for crossover

- Many GA problems also need special crossover rules.
- Since each genetic sequence contains all the cities in the travel, crossover is a swapping of travel order.
- Remember that crossover also needs to be efficient.

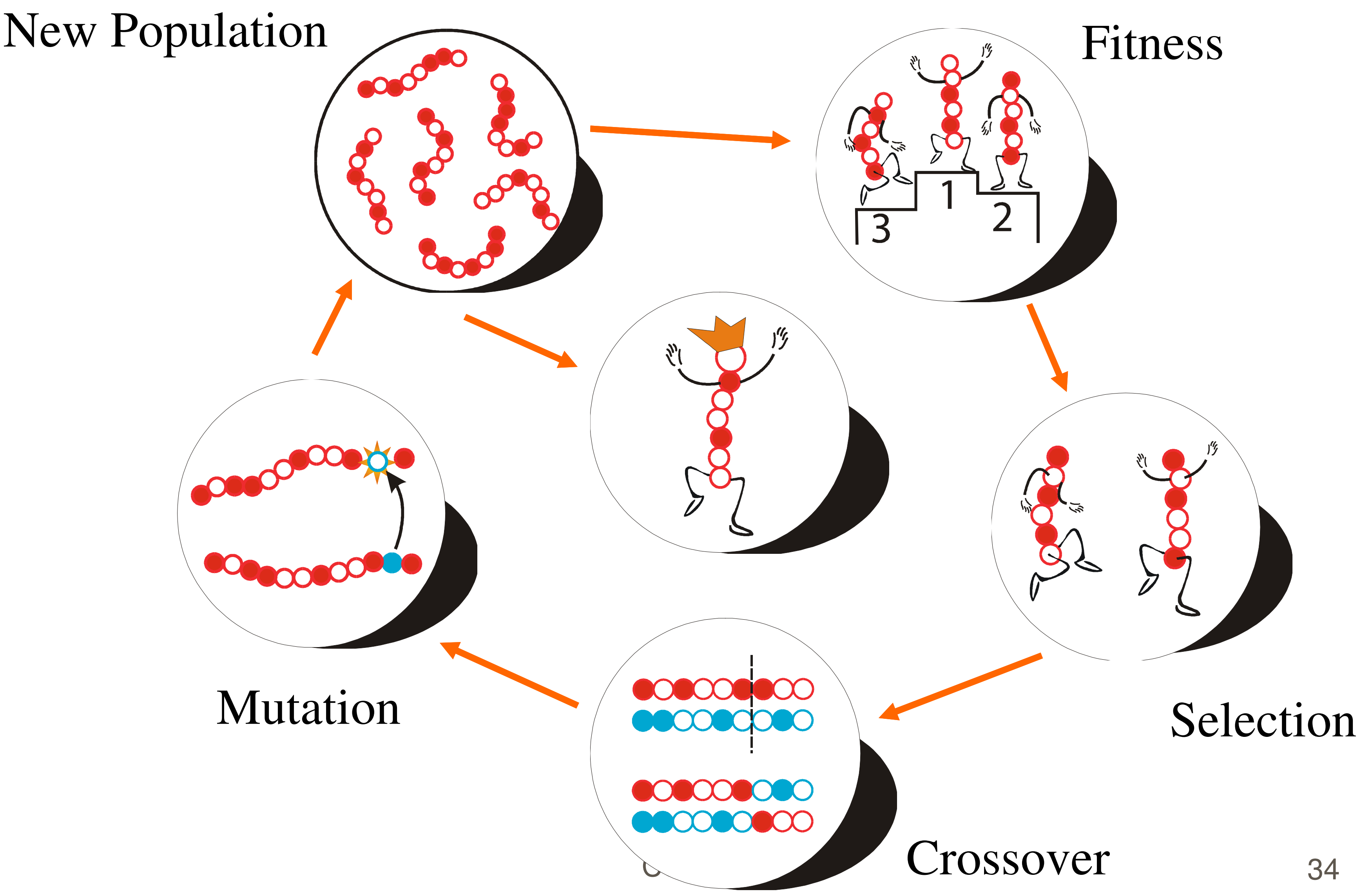


## What about local extrema?

- With just crossover breading, we are constrained to gene sequences which are a cross product of our current population.
- Introduce random effects into our population.
  - Mutation – Randomly twiddle the genes with some probability.
  - Cataclysm – Kill off n% of your population and create fresh new salespeople if it looks like you are reaching a local minimum.
  - Annealing of Mating Pairs – Accept the mating of suboptimal pairs with some probability.
  - Etc…



## In summation: The GA Cycle



## GA and TSP: the claims

- Can solve for over 3500 cities (still took over 1 CPU years).
  - Maybe holds the record.
- Will get within 2% of the optimal solution.
  - This means that it's not a solution per se, but is an approximation.

## GA Discussion

- We can apply the GA solution to any problem where the we can represent the problems solution (even very abstractly) as a string.
- We can create strings of:
  - Digits
  - Labels
  - Pointers
  - Code Blocks – This creates new programs from strung together blocks of code. The key is to make sure the code can run.
  - Whole Programs – Modules or complete programs can be strung together in a series. We can also re-arrange the linkages between programs.
- The last two are examples of Genetic Programming

## Things to consider

- How large is your population?
  - A large population will take more time to run (you have to test each member for fitness!).
  - A large population will cover more bases at once.
- How do you select your initial population?
  - You might create a population of approximate solutions. However, some approximations might start you in the wrong position with too much bias.
- How will you cross breed your population?
  - You want to cross breed and select for your best specimens.
    - Too strict: You will tend towards local minima
    - Too lax: Your problem will converge slower
- How will you mutate your population?
  - Too little: your problem will tend to get stuck in local minima
  - Too much: your population will fill with noise and not settle.

## GA is a good *no clue* approach to problem solving

- GA is superb if:
  - Your space is loaded with lots of weird bumps and local minima.
    - GA tends to spread out and test a larger subset of your space than many other types of learning/optimization algorithms.
  - You don't quite understand the underlying *process* of your problem space.
    - NO I DONT: What makes the stock market work??? Don't know? Me neither! Stock market prediction might thus be good for a GA.
    - YES I DO: Want to make a program to predict people's height from personality factors? This might be a Gaussian process and a good candidate for statistical methods which are more efficient.
  - You have lots of processors
    - GA's parallelize very easily!

## Summary

- Best-first search = general search, where the minimum-cost nodes (according to some measure) are expanded first.

- Greedy search = best-first with the estimated cost to reach the goal as a heuristic measure.
  - Generally faster than uninformed search
  - not optimal
  - not complete.

- A* search = best-first with measure = path cost so far + estimated path cost to goal.
  - combines advantages of uniform-cost and greedy searches
  - complete, optimal and optimally efficient
  - space complexity still exponential

## Summary

- Time complexity of heuristic algorithms depend on quality of heuristic function. Good heuristics can sometimes be constructed by examining the problem definition or by generalizing from experience with the problem class.
- Iterative improvement algorithms keep only a single state in memory – best example is hill-climbing (greedy, local search)
- Greedy Hill Climbing search can get stuck in local extrema;
- Simulated annealing provides a way to escape local extrema, and is complete and optimal given a slow enough cooling schedule.
- Genetic Algorithms provide another alternative – you can combine it with mutation and annealing to escape local optima

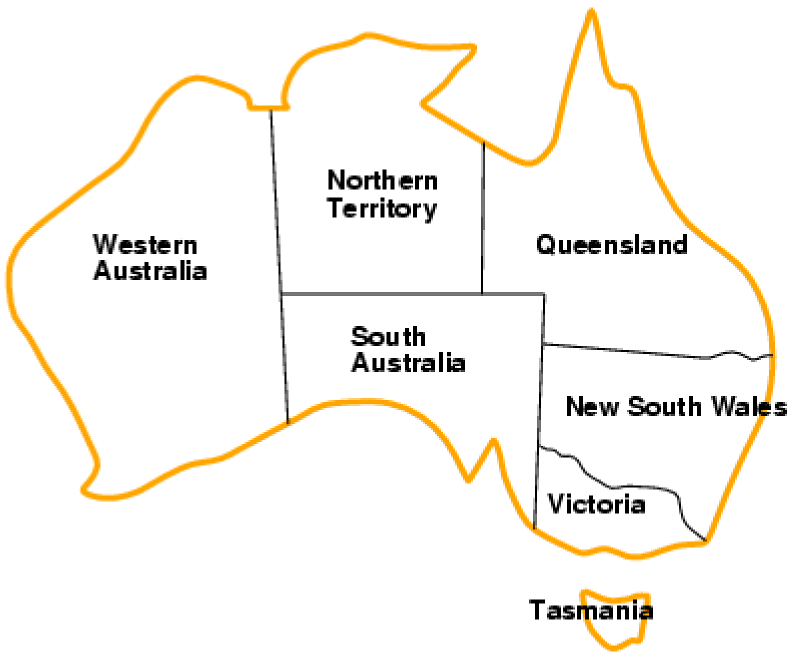
## Constraint satisfaction problems (CSPs)

- Standard search problem:
  - state is a "black box" – any data structure that supports successor function, heuristic function, and goal test
- 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

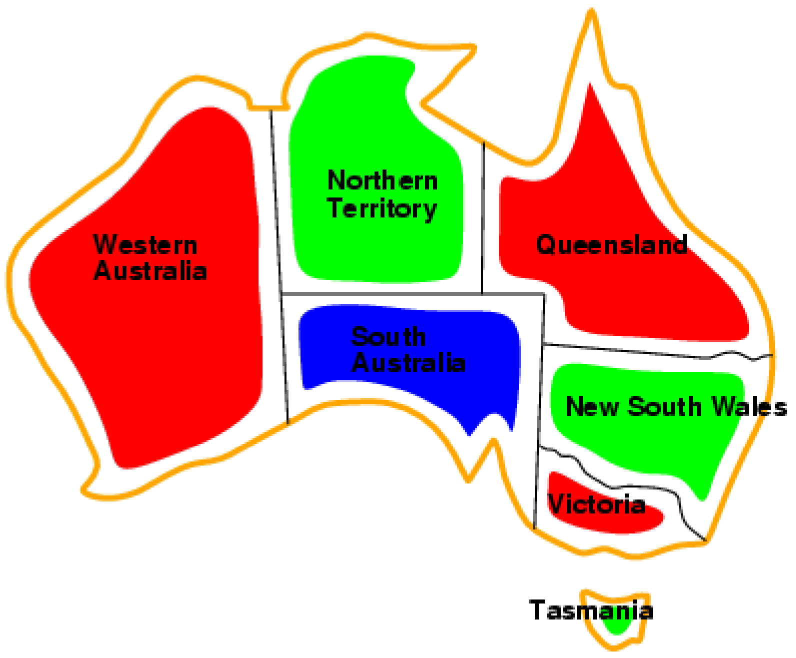
- Allows useful general-purpose algorithms with more power than standard search algorithms to generate solutions

## 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 ≠ NT, or (WA,NT) in {(red,green),(red,blue),(green,red), (green,blue),(blue,red),(blue,green)}

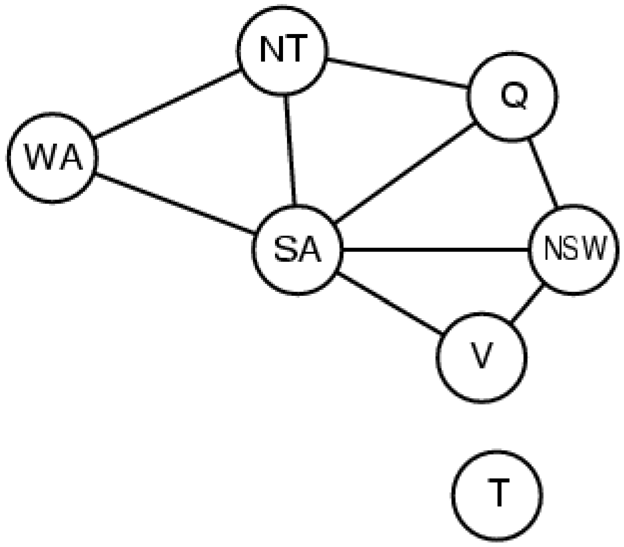
## Example: Map-Coloring



- Solutions are complete and consistent assignments of values to the variables
- e.g., WA = red, NT = green,Q = red,NSW = green,V = red,SA = blue,T = green

## Constraint graph

- Binary CSP: each constraint relates two variables
- Constraint graph: nodes are variables, arcs are constraints



## Varieties of CSPs

- Discrete variables
  - finite domains:
    - *$n$* variables, domain size *$d$* → *$O(d^n)$* 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., $StartJob_1 + 5 \leq StartJob_3$

- Continuous variables
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by linear programming

## Varieties of constraints

- Unary constraints involve a single variable,
  - e.g., SA ≠ green
  - 
- Binary constraints involve pairs of variables,
  - e.g., SA ≠ WA
  - 
- Higher-order constraints involve 3 or more variables,
  - e.g., cryptarithmetic column constraints
  -

## Standard search formulation (incremental)

Let's start with the straightforward 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

- 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

## Backtracking search

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

[ WA = red then NT = green ] same as [ NT = green then WA = red ]

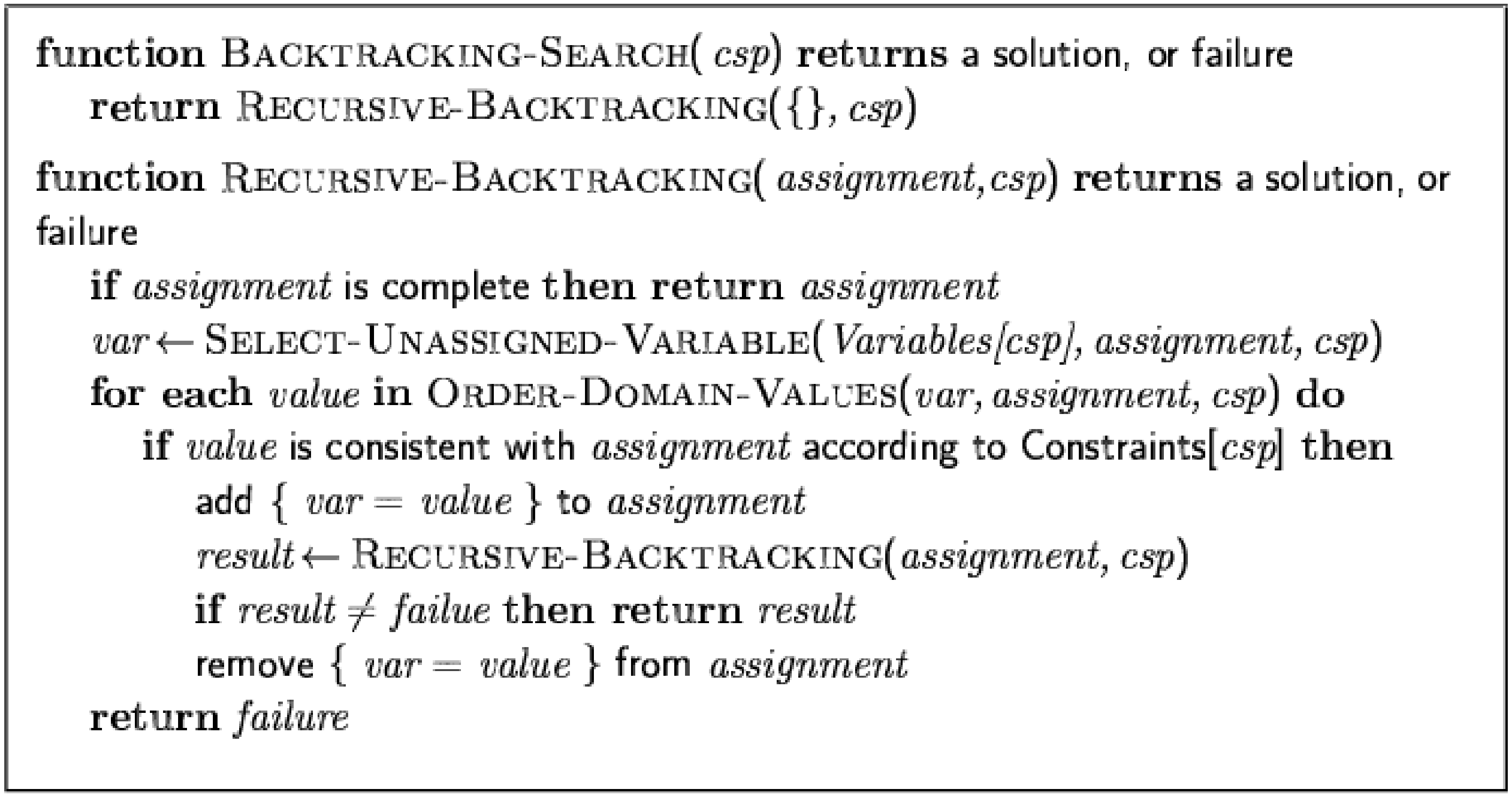
- Only need to consider assignments to a single variable at each node
  → b = d and there are \$d^n\$ leaves

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

- Backtracking search is the basic uninformed algorithm for CSPs

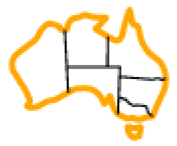
- Can solve *n*-queens for *n* ≈ 25

## Backtracking search

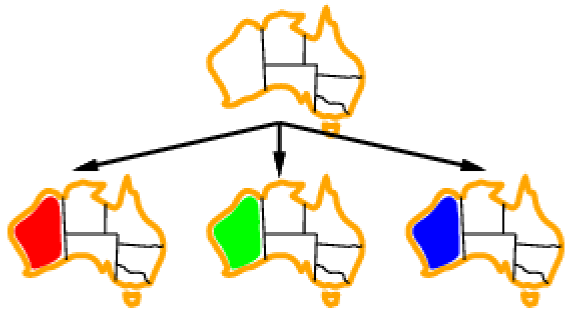


```
function BACKTRACKING-SEARCH( csp) returns a solution, or failure
    return RECURSIVE-BACKTRACKING({}, csp)
function RECURSIVE-BACKTRACKING( assignment, csp) returns a solution, or
failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE( Variables[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to Constraints[csp] then
            add { var = value } to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failue then return result
            remove { var = value } from assignment
    return failure
```

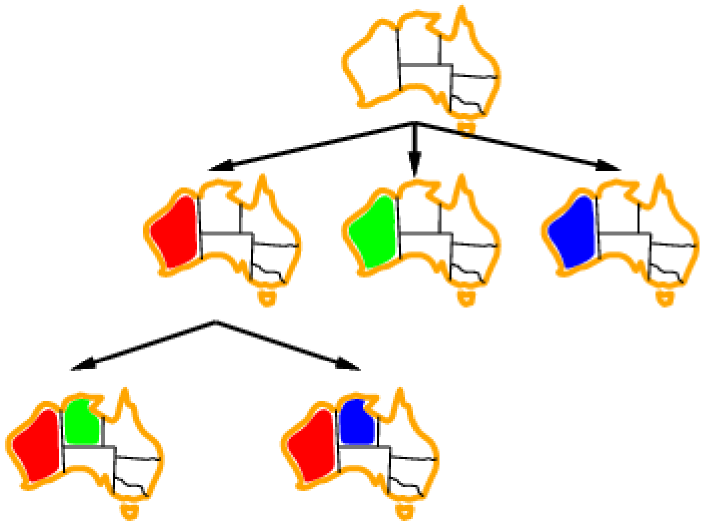
## Backtracking example



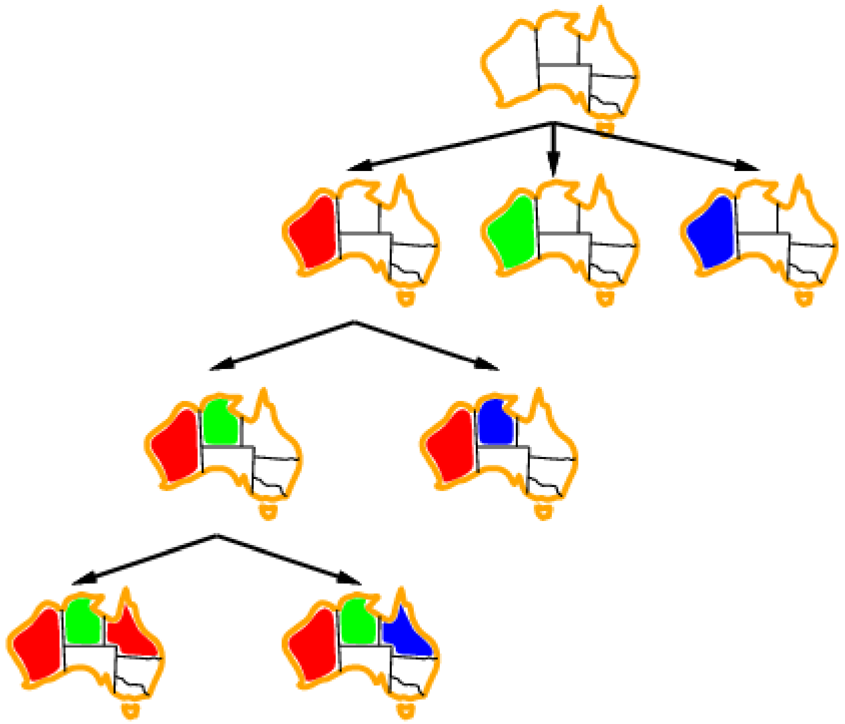
## Backtracking example



## Backtracking example



## Backtracking example

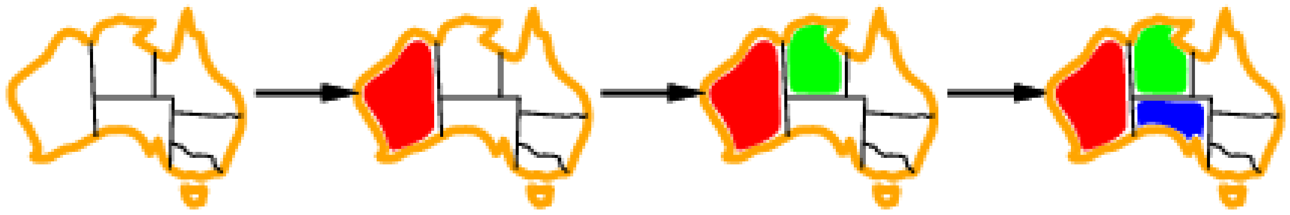


## Most constrained variable

- Most constrained variable:

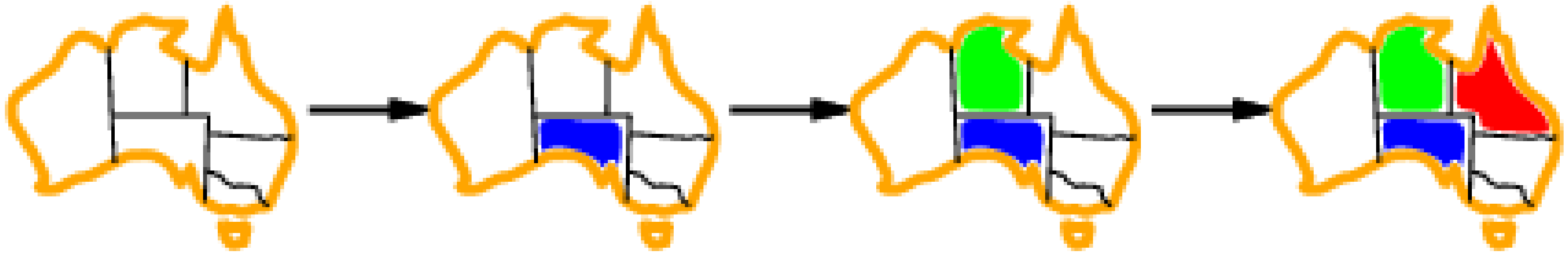
  choose the variable with the fewest legal values

- a.k.a. minimum remaining values (MRV) heuristic



## Most constraining variable

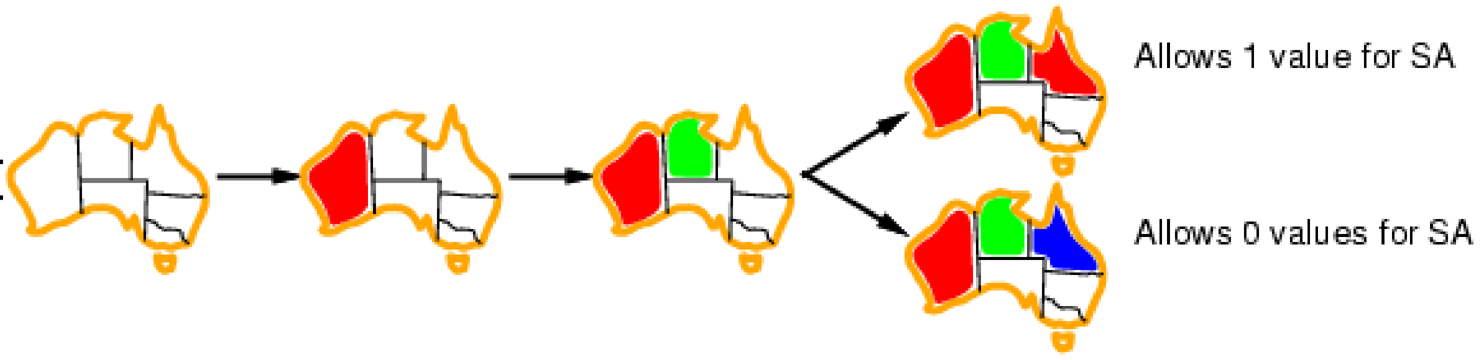
- Tie-breaker among most constrained variables
- Most constraining variable:
  - choose the variable with the most constraints on remaining variables



## Least constraining value

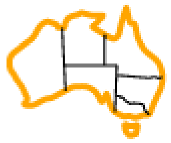
- Given a variable, choose the least constraining value:
  - the one that rules out the fewest values in the remaining variables

- C



## Forward checking

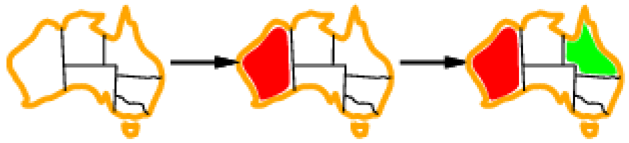
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values
  -



| WA | NT | Q | NSW | V | SA | T |
|---|---|---|---|---|---|---|
| 🟥🟩🟦 | 🟥🟩🟦 | 🟥🟩🟦 | 🟥🟩🟦 | 🟥🟩🟦 | 🟥🟩🟦 | 🟥🟩🟦 |

## Forward checking

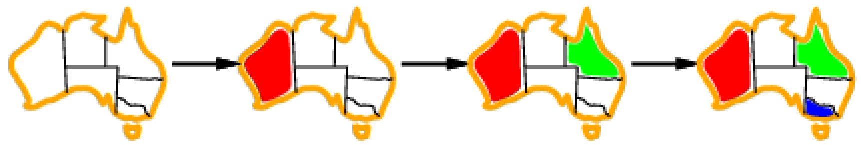
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values
  -



| WA | NT | Q | NSW | V | SA | T |
|---|---|---|---|---|---|---|
| 🟥🟩🟦 | 🟥🟩🟦 | 🟥🟩🟦 | 🟥🟩🟦 | 🟥🟩🟦 | 🟥🟩🟦 | 🟥🟩🟦 |
| 🟥 | 🟩🟦 | 🟥🟩🟦 | 🟥🟩🟦 | 🟥🟩🟦 | 🟩🟦 | 🟥🟩🟦 |
| 🟥 | 🟦 | 🟩 | 🟥🟦 | 🟥🟩🟦 | 🟦 | 🟥🟩🟦 |

## Forward checking

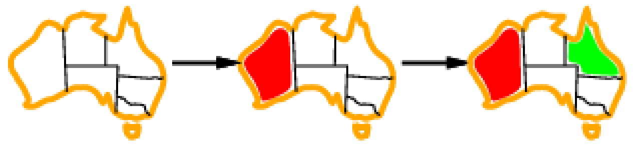
- Idea:
    - Keep track of remaining legal values for unassigned variables
    - Terminate search when any variable has no legal values
    -

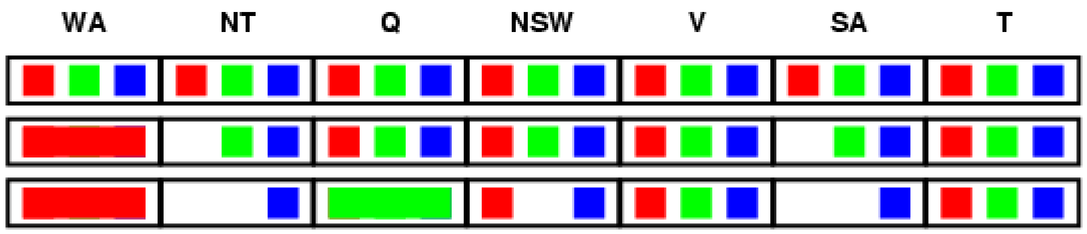


| WA | NT | Q | NSW | V | SA | T |
|---|---|---|---|---|---|---|
| 🟥🟩🟦 | 🟥🟩🟦 | 🟥🟩🟦 | 🟥🟩🟦 | 🟥🟩🟦 | 🟥🟩🟦 | 🟥🟩🟦 |
| 🟥 | 🟩🟦 | 🟥🟩🟦 | 🟥🟩🟦 | 🟥🟩🟦 | 🟩🟦 | 🟥🟩🟦 |
| 🟥 | 🟦 | 🟩 | 🟥🟦 | 🟥🟩🟦 | 🟦 | 🟥🟩🟦 |
| 🟥 | 🟦 | 🟩 | 🟥 | 🟦 |  | 🟥🟩🟦 |

## 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