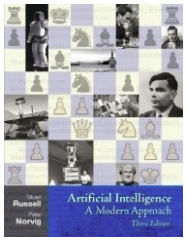
Outline

- Best-first search
  - Greedy best-first search
  - A* search
  - Heuristics
- Local search algorithms
  - Hill-climbing search
  - Beam search
  - Simulated annealing search
  - Genetic algorithms
- Constraint Satisfaction Problems
  - Constraints
  - Constraint Propagation
  - Backtracking Search
  - Local Search



Many slides based on Russell & Norvig's slides Artificial Intelligence: A Modern Approach

So far:

methods that systematically explore the search space, possibly using principled pruning (e.g., A*)

Current best such algorithm can handle search spaces of up to 10^{100} states / around 500 binary variables ("ballpark" number only!)

What if we have much larger search spaces?

Search spaces for some real-world problems may be much larger e.g. $10^{30,000}$ states as in certain reasoning and planning tasks.

A completely different kind of method is called for --- non-systematic:

Local search
(sometimes called: Iterative Improvement Methods)

Local Search Algorithms
(sometimes called: Iterative Improvement Methods)

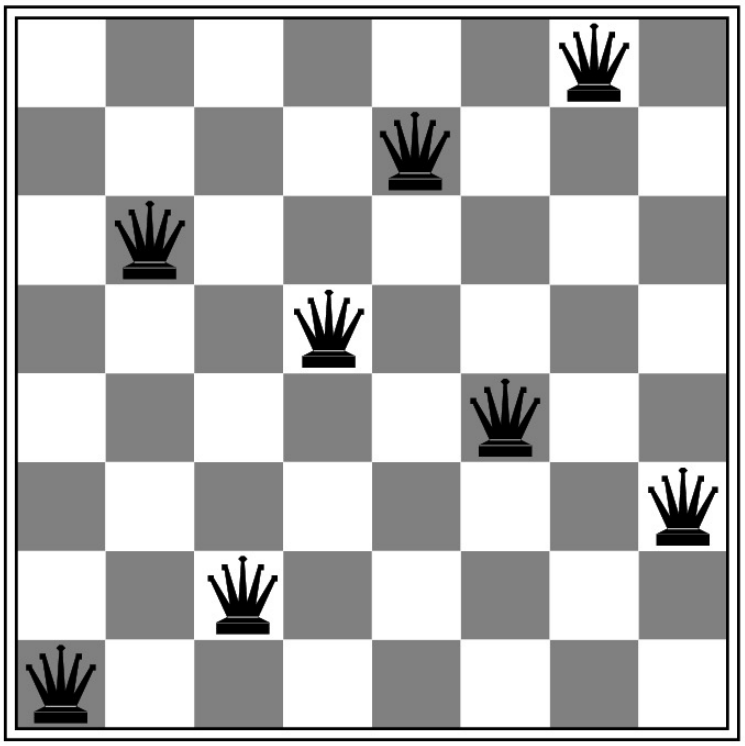
- In many optimization problems, the path to the goal is irrelevant
  - the goal state itself is the solution

  - State space:
    - set of "complete" configurations
  - Goal:
    - Find a configuration that satisfies all constraints

- Examples:
  - n-queens problem, travelling salesman, …

- In such cases, we can use local search algorithms

N-queens Problem



We do not want the path to the goal. The solution is all what matters.

Local Search

- **Approach**
  - keep a single "current" state (or a fixed number of them)
  - try to improve it by maximizing a heuristic evaluation
  - using only **„local“ improvements**
    - i.e., only modifies the current state(s)
  - paths are typically not remembered
  - similar to solving a puzzle by hand
    - e.g., 8-puzzle, Rubik's cube

- Advantages
  - uses very little memory
  - often quickly finds solutions in large or infinite state spaces

- Disadvantages
  - no guarantees for completeness or optimality

Optimization Problems

- Goal:
  - optimize some evaluation function (objective function)
- there is no goal state, and no path costs
  - hence A* and other algorithms we have discussed so far are not applicable

Named like this because it defines the „objective“ that has to be met

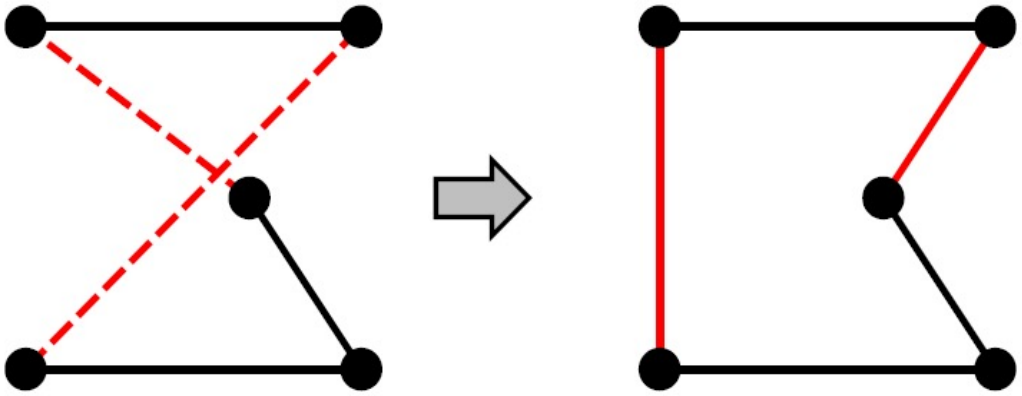
- Example:
  - Darwinian evolution and survival of the fittest may be regarded as an optimization process

Algorithm design considerations

- How do you represent your problem?
- What is a “complete state”?
- What is your objective function?
  - How do you measure cost or value of a state?
- What is a “neighbor” of a state?
  - Or, what is a “step” from one state to another?
  - How can you compute a neighbor or a step?
- Are there any constraints you can exploit?

Example: Travelling Salesman Problem

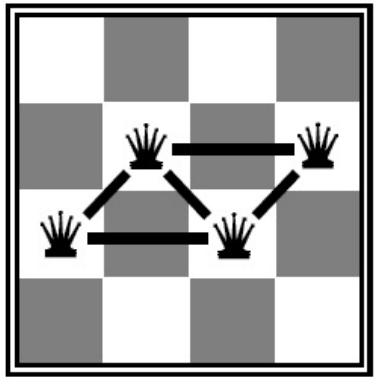
- Basic Idea:
  - Start with a complete tour
  - perform pairwise exchanges



  - variants of this approach get within 1% of an optimal solution very quickly with thousands of cities

Example: n-Queens Problem

- Basic Idea:
  - move a queen so that it reduces the number of conflicts

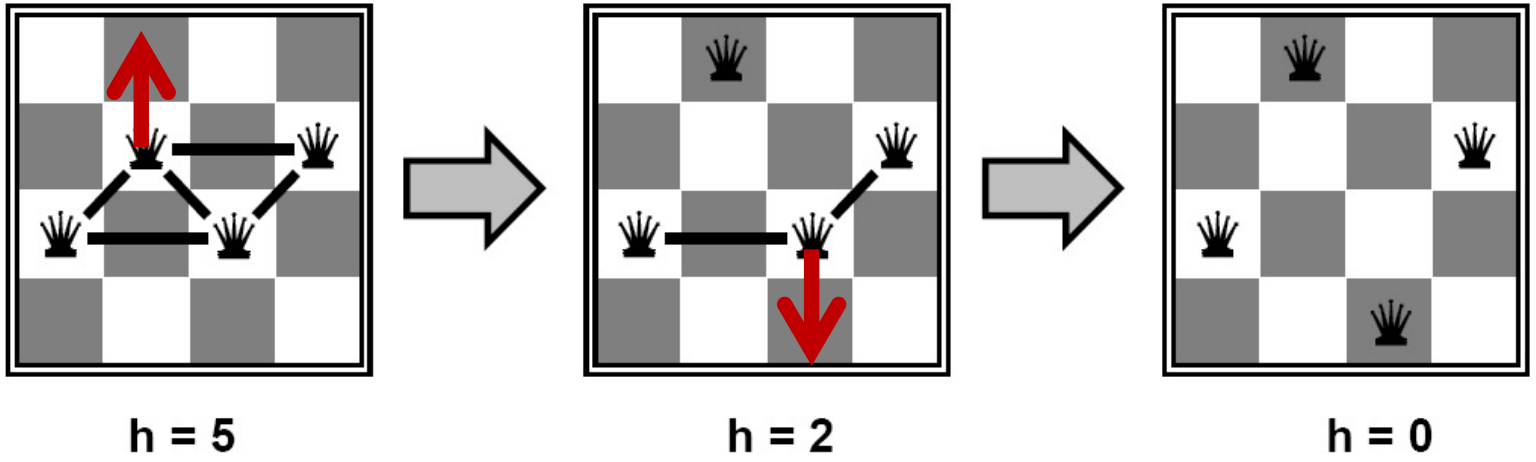


h = 5

States: 4 queens in 4 columns (256 states)
Neighborhood Operators: move queen in column
Evaluation / Optimization function: h(n) = number of attacks / "conflicts"
Goal test: no attacks, i.e., h(G) = 0

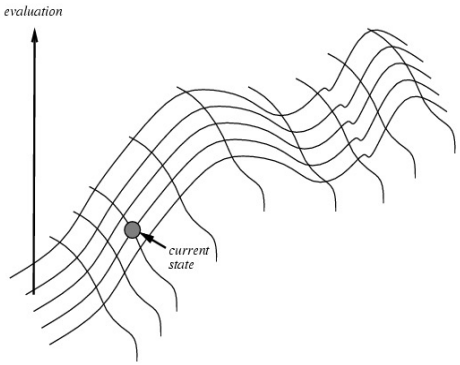
Example: n-Queens Problem

- Basic Idea:
  - move a queen so that it reduces the number of conflicts



- Local search: Because we only consider local changes to the state at each step. We generally make sure that series of local changes can reach all possible states.
- almost always solves n-queens problems almost instantaneously for very large n (e.g., n = 1,000,000)

Hill-climbing search



Algorithm:

- expand the current state (generate all neighbors)
- move to the one with the highest evaluation
- until the evaluation goes down

```
function HILL-CLIMBING( problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

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

Hill-climbing search
(aka Greedy Local Search)

- Algorithm:
  - expand the current state (generate all neighbors)
  - move to the one with the highest evaluation
  - until the evaluation goes down

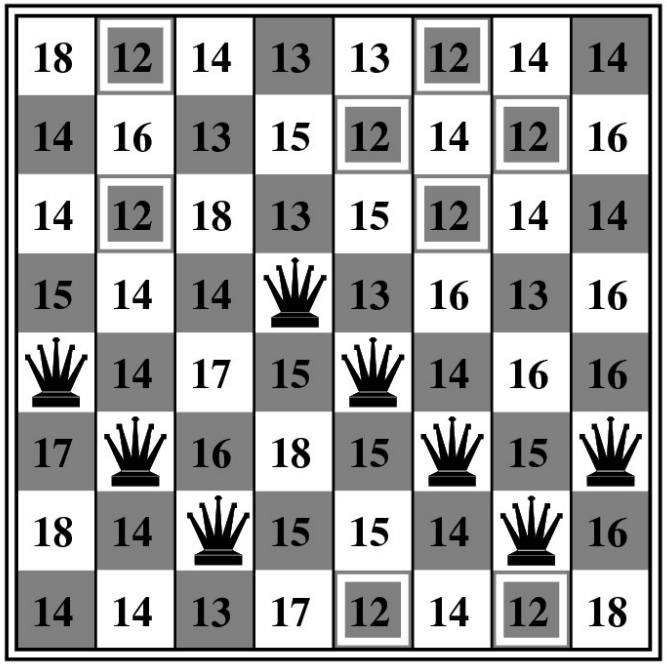
- Main Problem: Local Optima
  - the algorithm will stop as soon as is at the top of a hill
  - but it is actually looking for a mountain peak

> *"Like climbing Mount Everest in thick fog with amnesia"*

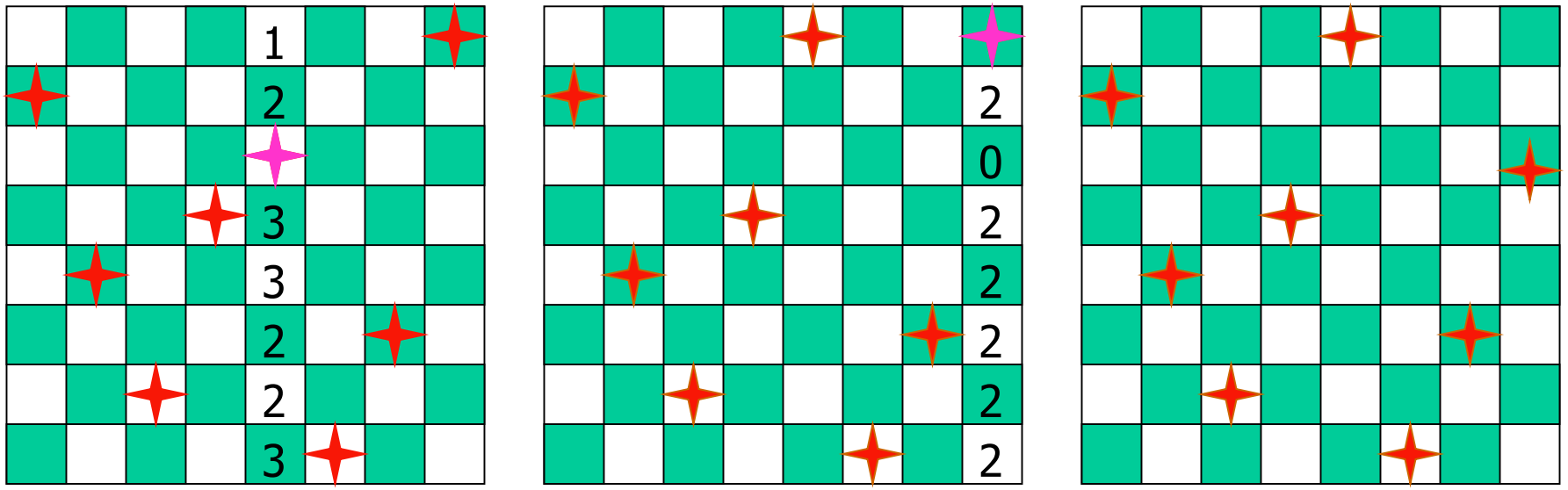
- Other problems:
  - ridges
  - plateaux
  - shoulders

Example: 8-Queens Problem

- Heuristic h:
  - number of pairs of queens that attach each other
- Example state: $h = 17$



Example: 8-Queens Problem



Representation: 8 integer variables giving positions of 8 queens in columns
(e.g. <2, 5, 7, 4, 3, 8, 6, 1>)

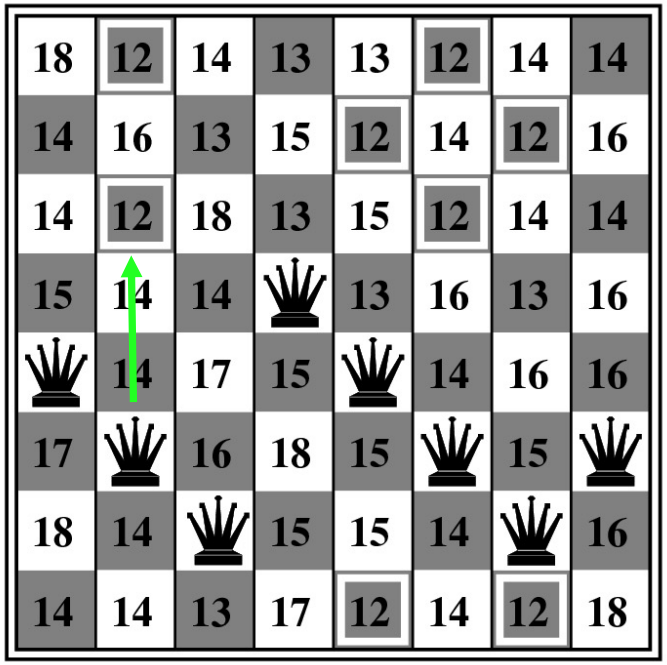
Pick initial complete assignment (at random)
Repeat

- **Pick a conflicted variable var (at random)**
- **Set the new value of var to minimize the number of conflicts**
- **If the new assignment is not conflicting then return it**

(Min-conflicts heuristics) → **Inspired GSAT and Walksat (see later)**

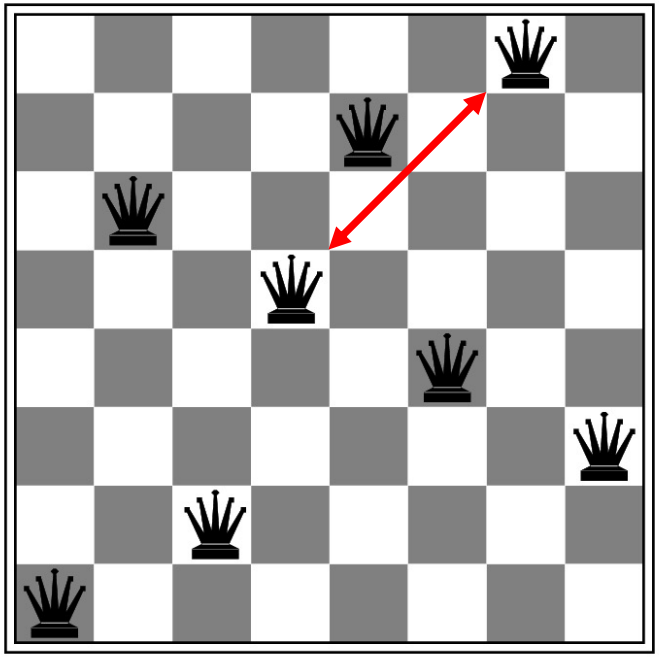
Example: 8-Queens Problem

- Heuristic h:
  - number of pairs of queens that attack each other, if we move a queen in ist column to that square
- Example state: $h = 17$



- Best Neighbor(s): $h = 12$

- Local optimum with $h = 1$



  - no queen can move without increasing the number of attacked pairs
  - What can you do to get out of this local minimum?

Randomized Hill-Climbing Variants

- Random Restart Hill-Climbing
  - Different initial positions result in different local optima

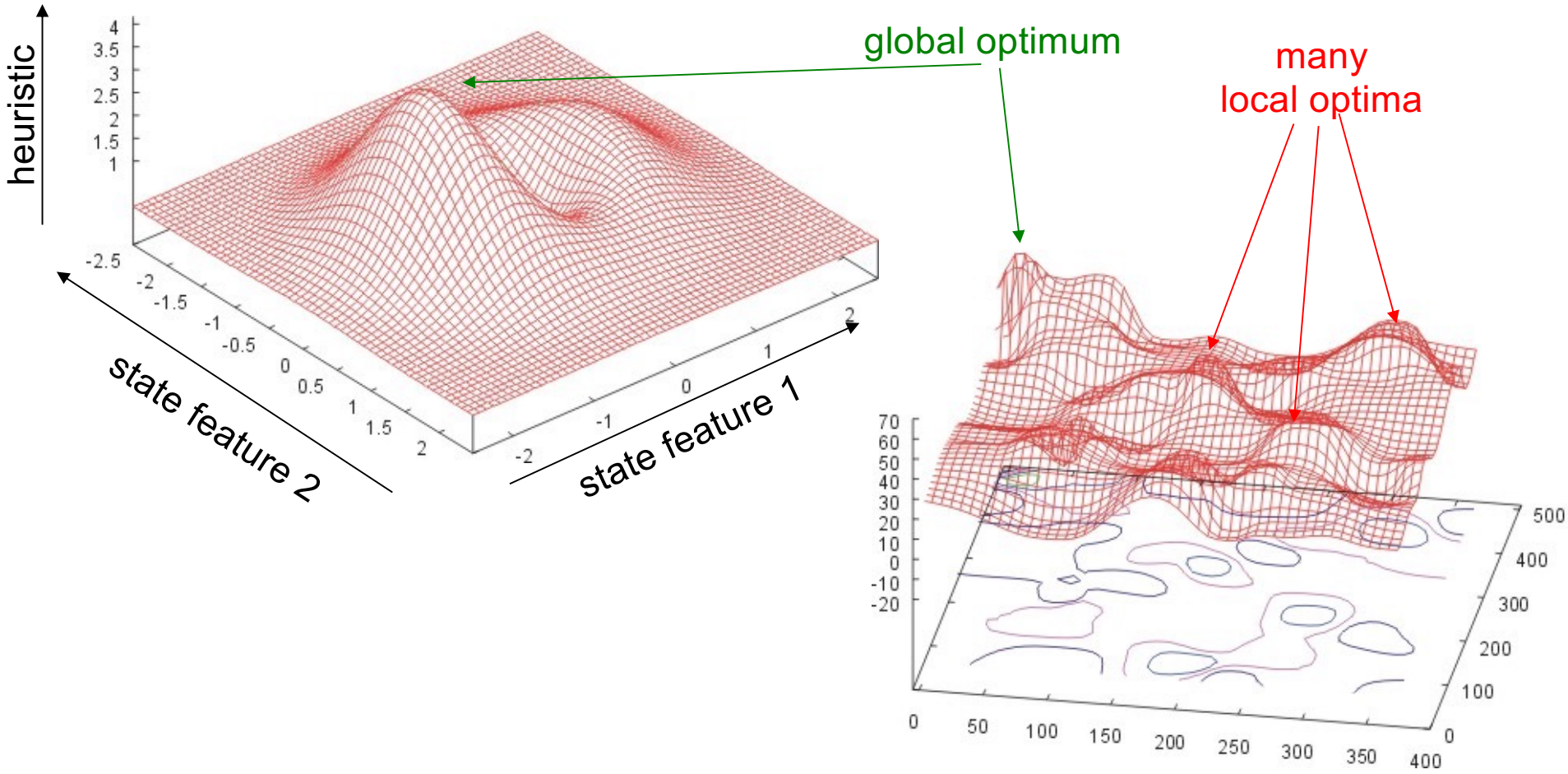
  → make several iterations with different starting positions
- Example:
  - for 8-queens problem the probability that hill-climbing succeeds from a randomly selected starting position is ≈ 0.14

  → a solution should be found after about $1/0.14 \approx 7$ iterations of hill-climbing

- Stochastic Hill-Climbing
  - select the successor node ramdomly
  - better nodes have a higher probability of being selected

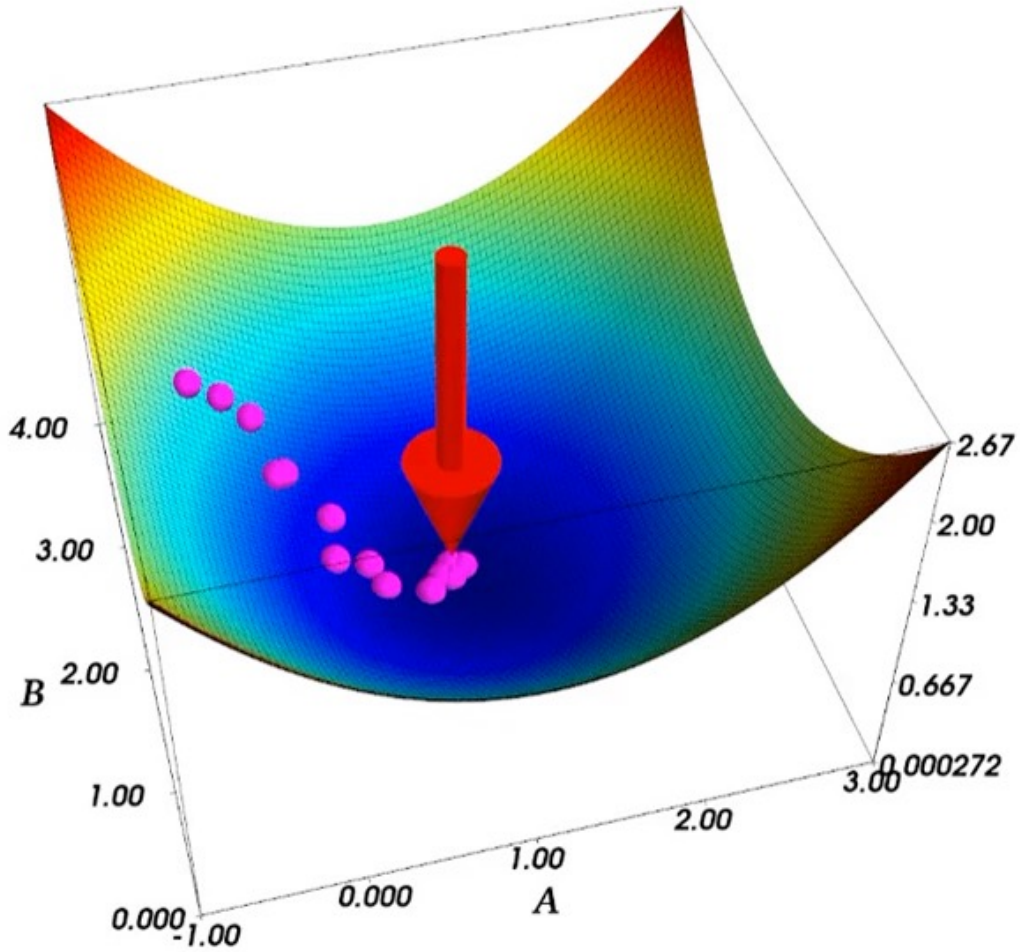
Multi-Dimensional State-Landscape

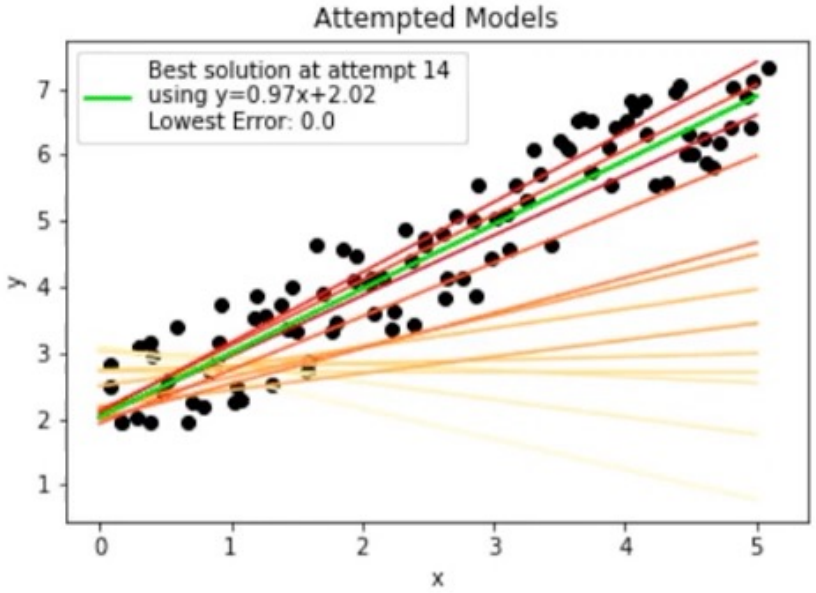
- States may be refined in multiple ways
  → similarity along various dimensions

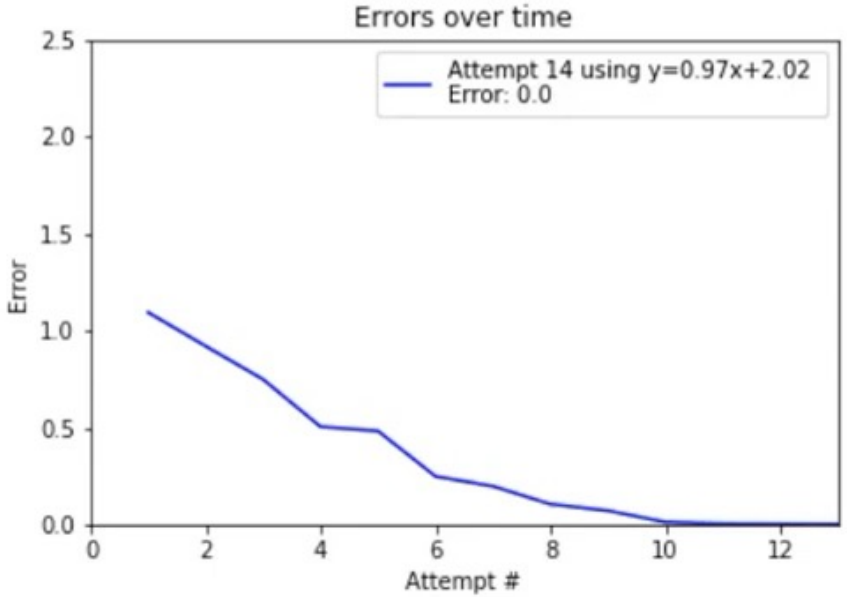


Optimization Problems

Frederik Hardervig
https://www.youtube.com/watch?v=z3qOOJl-VSU



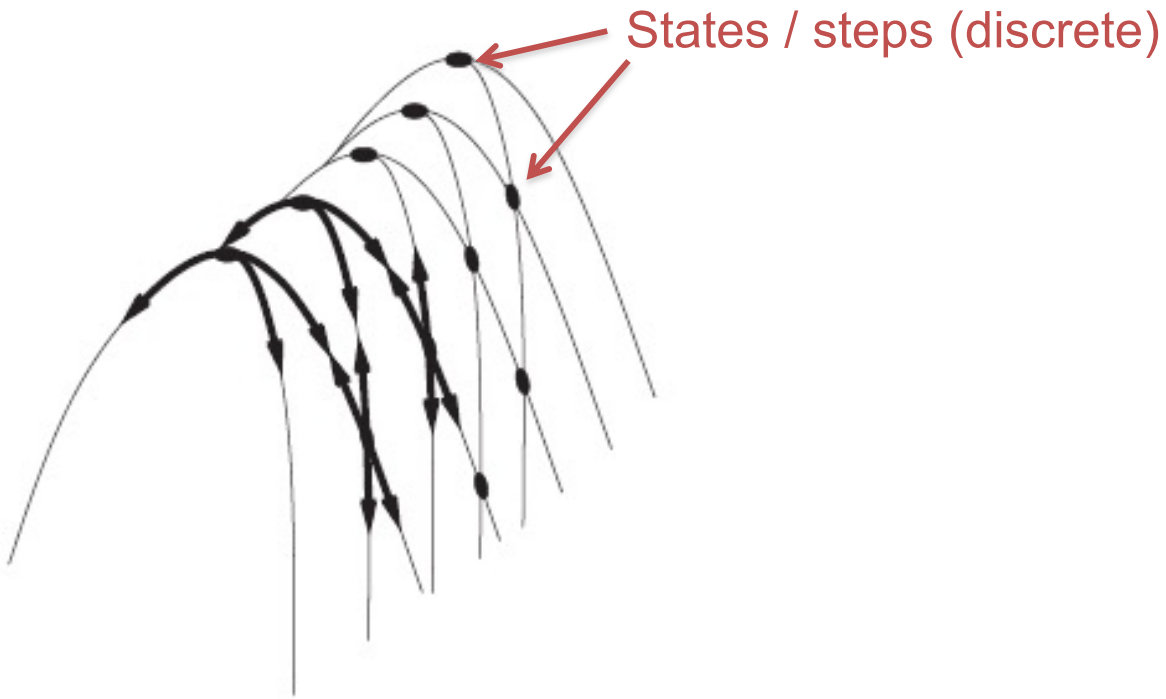




Ridge Problem

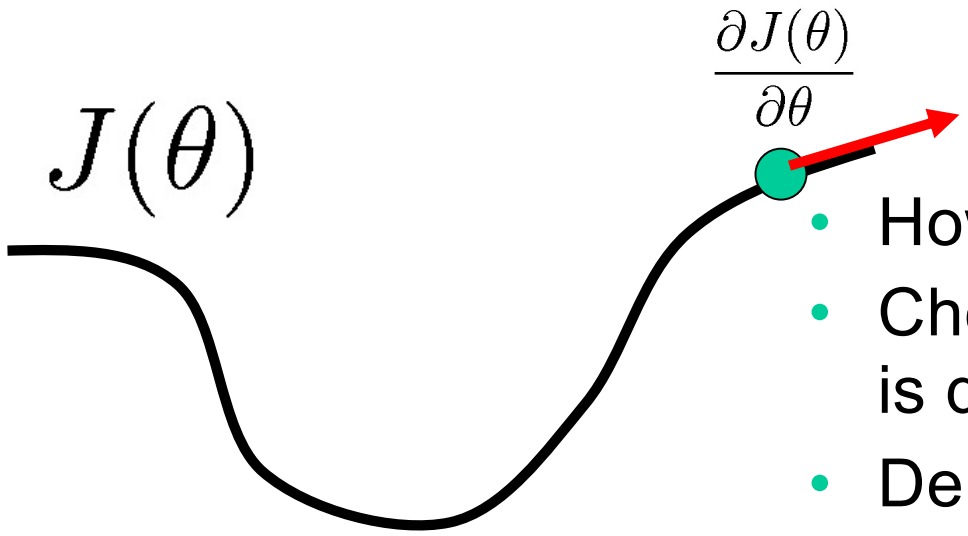
- Every neighbor appears to be downhill
  - But, search space has an uphill (just not in neighbors)

Ridge:
Fold a piece of paper and hold it tilted up at an unfavorable angle to every possible search space step. Every step leads downhill; but the ridge leads uphill.



Gradient descent

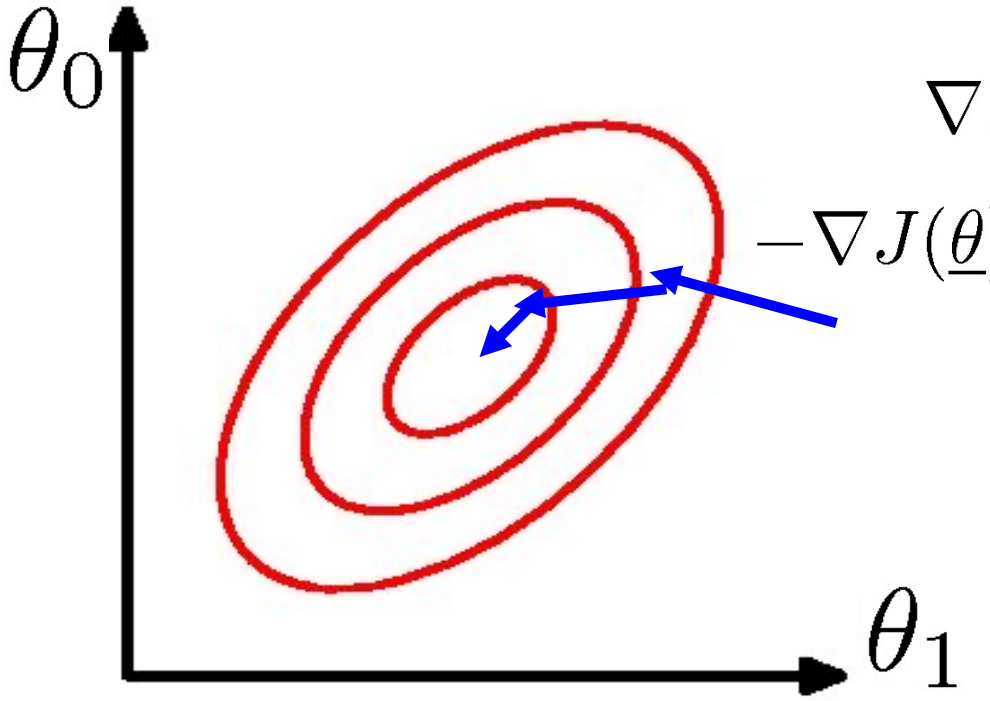
- Hill-climbing in continuous state spaces
- Denote “state” as μ; cost as J(μ)



- How to change μ to improve J(μ)?
- Choose a direction in which J(μ) is decreasing
- Derivative $\frac{\partial J(\theta)}{\partial \theta}$
  - Positive => increasing
  - Negative => decreasing

Gradient descent

Hill-climbing in continuous spaces



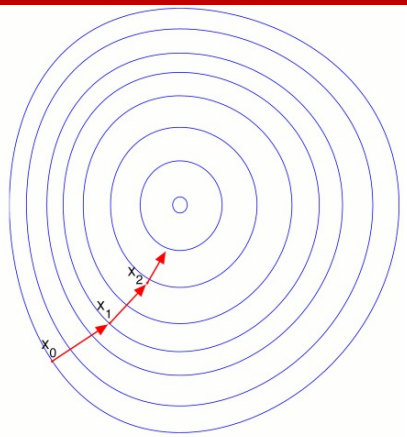
- Gradient vector

$$\nabla J(\underline{\theta}) = \left[\frac{\partial J(\underline{\theta})}{\partial \theta_0} \quad \frac{\partial J(\underline{\theta})}{\partial \theta_1} \quad \dots \right]$$

- Indicates direction of steepest ascent

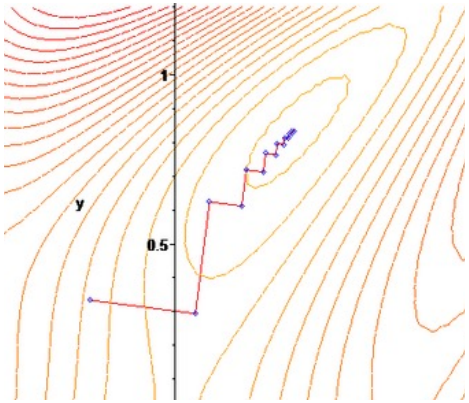
  (negative = steepest descent)

Gradient descent



Hill-climbing in continuous spaces

Gradient = the most direct direction up-hill in the objective (cost) function, so its negative minimizes the cost function.



* Assume we have some cost-function: $J(x_1, x_2, \ldots, x_n)$ and we want minimize over continuous variables $x_1, x_2, .., x_n$

1. Compute the *gradient :* $\frac{\partial}{\partial x_i} J(x_1, \ldots, x_n) \qquad \forall i$
2. Take a small step downhill in the direction of the gradient:

$$x_i' = x_i - \lambda \frac{\partial}{\partial x_i} J(x_1, \ldots, x_n)$$

3. Check if $J(x_1', \ldots, x_n') < J(x_1, \ldots, x_n)$
4. If true then accept move, if not "reject".
5. Repeat.

(or, Armijo rule, etc.)

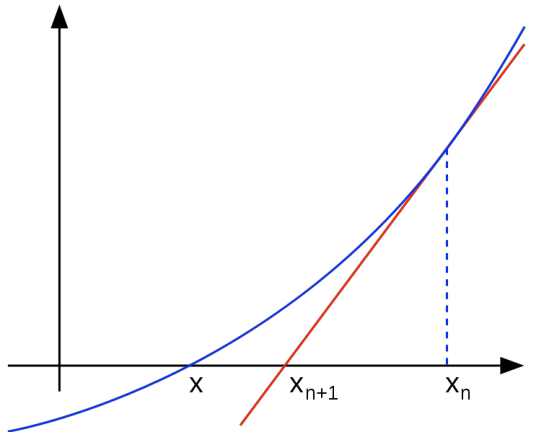
(decrease step size, etc.)

Gradient descent

Hill-climbing in continuous spaces

- How do I determine the gradient?
  - Derive formula using multivariate calculus.
  - Ask a mathematician or a domain expert.
  - Do a literature search.
- Variations of gradient descent can improve performance for this or that special case.
  - See Numerical Recipes in C (and in other languages) by Press, Teukolsky, Vetterling, and Flannery.
  - Simulated Annealing, Linear Programming too
- Works well in smooth spaces; poorly in rough.

Newton's method



- Want to find the roots of f(x)
  - "Root": value of x for which f(x)=0
- Initialize to *some* point x
- Compute the tangent at x & compute where it crosses x-axis

$$\nabla f(x) = \frac{0 - f(x)}{x' - x} \quad \Rightarrow \quad x' = x - \frac{f(x)}{\nabla f(x)}$$

- Optimization: find roots of df(x)

$$\nabla\nabla f(x) = \frac{0 - \nabla f(x)}{x' - x} \quad \Rightarrow \quad x' = x - \frac{\nabla f(x)}{\nabla\nabla f(x)}$$

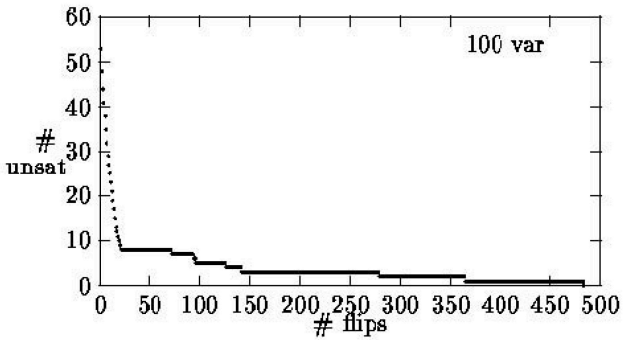
("Step size" , = 1/ddf ; inverse curvature)

- Does not always converge; sometimes unstable
- If converges, usually very fast
- Works well for smooth, non-pathological functions, linearization accurate
- Works poorly for wiggly, ill-behaved functions

(Multivariate:
d f(x) = gradient vector
d^2 f(x) = matrix of 2^{nd} derivatives
a/b = a b^{-1}, matrix inverse)

Another example: Greedy SAT

- Task: Find a satifying configuration I of a propositional formula **Δ**
- Hill-climbing for SAT in a discrete space



auxiliary functions:

- violated(Δ, I): number of clauses in Δ not satisfied by I
- flip(I, v): assignment that results from I when changing the valuation of proposition v

function GSAT(Δ):

repeat *max-tries* **times**:
    $I :=$ a random assignment
    repeat *max-flips* **times**:
        if $I \models \Delta$:
            return I
        $V_{\text{greedy}} :=$ the set of variables v occurring in Δ
                for which violated$(\Delta, \text{flip}(I, v))$ is minimal
        randomly select $v \in V_{\text{greedy}}$
        $I := \text{flip}(I, v)$
return no solution found

Satisfiability (SAT) Problem

SAT Problem: given a propositional formula F in CNF, return a model (solution) to F or prove that none exists

Example: $F = (a + b)\ (a + \neg c)\ (\neg a + c)$

F has 3 models:

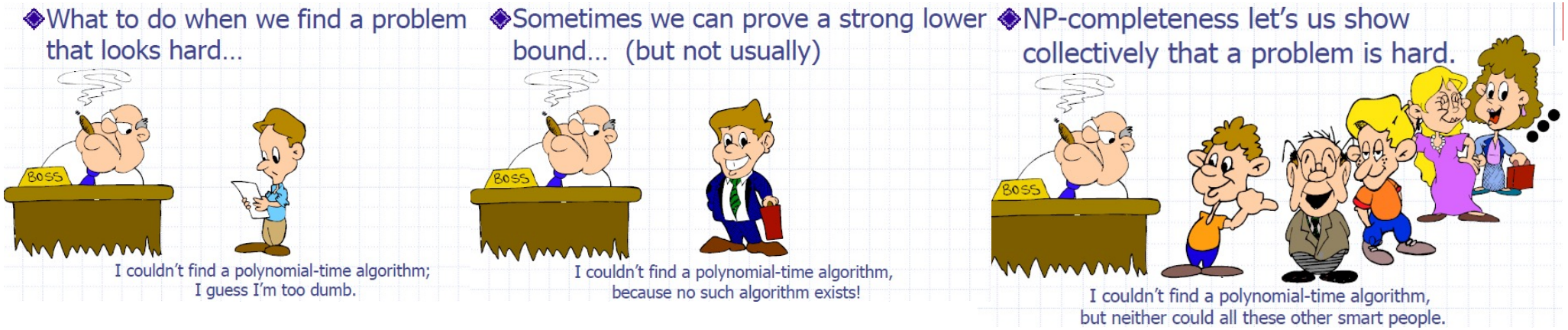
- $M_1 = \{a=0, b=1, c=0\}$
- $M_2 = \{a=1, b=0, c=1\}$
- $M_3 = \{a=1, b=1, c=1\}$

A SAT solver will return either of the 3 models

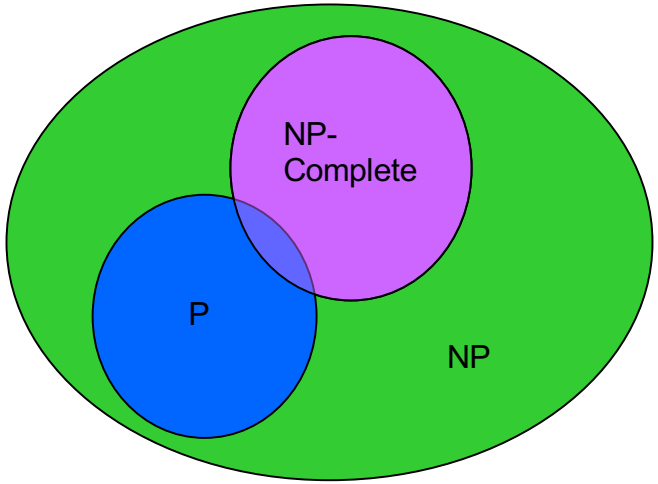
Example: $F = (a)\ (\neg a)$

F is not satisfiable

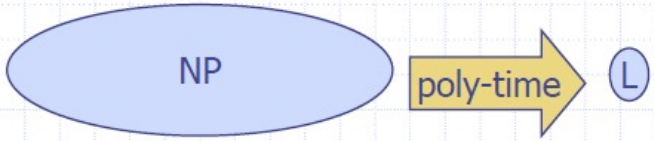
- SAT has myriads of applications

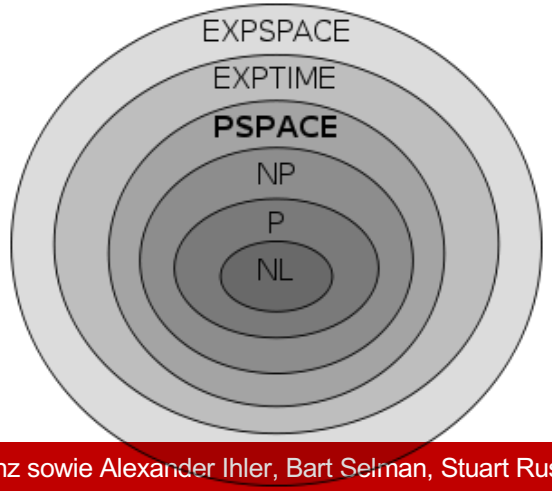


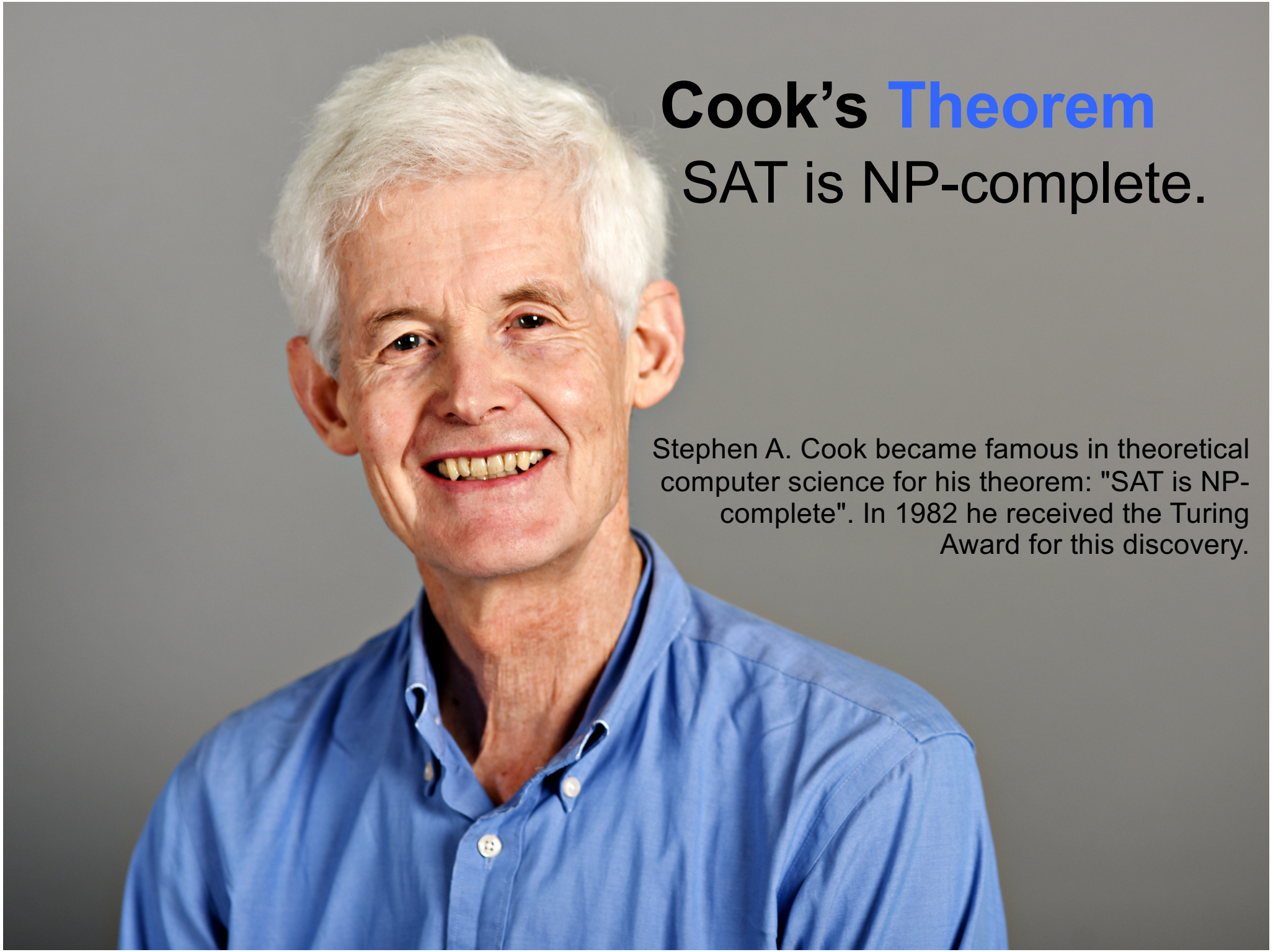
- P = { L | L is accepted by a deterministic Turing Machine in polynomial time }
- NP = { L | L is accepted by a non-deterministic Turing Machine in polynomial time }



- A problem (language) L is **NP-hard** if every problem in NP can be reduced to L in polynomial time.
- That is, for each language M in NP, we can take an input x for M, **transform** it in polynomial time to an input x' for L such that x is in M if and only if x' is in L.
- L is **NP-complete** if it's in NP and is NP-hard.





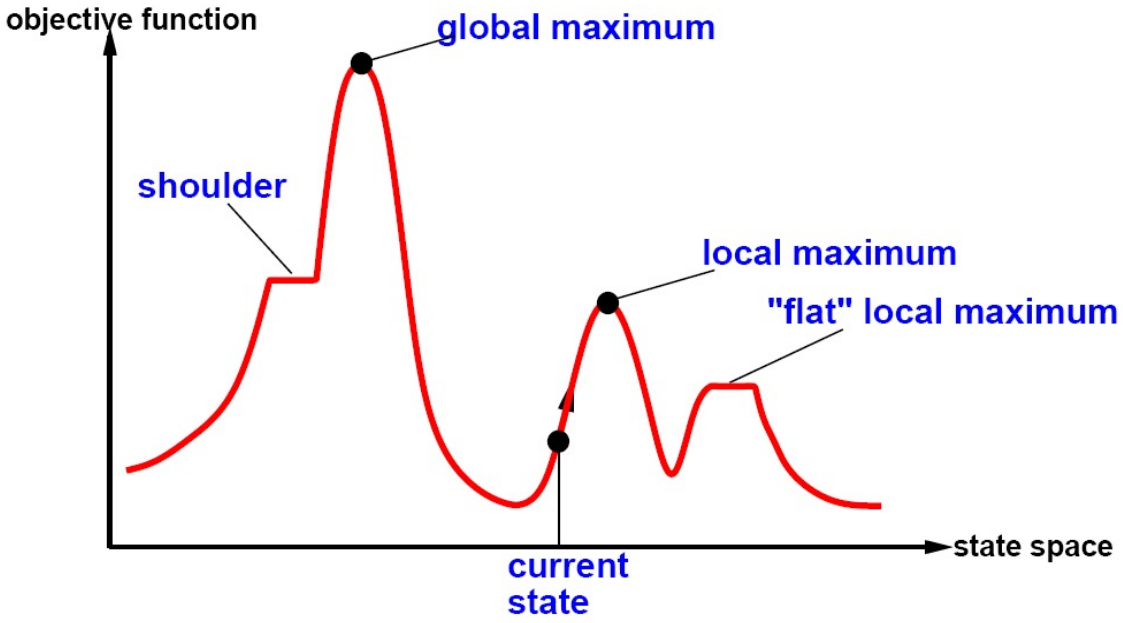

Cook's Theorem
SAT is NP-complete.
Stephen A. Cook became famous in theoretical computer science for his theorem: "SAT is NP-complete". In 1982 he received the Turing Award for this discovery.

State Space Landscape

- state-space landscape
  - location: states
  - elevation: heuristic value (objective function)
- Assumption:
  - states have some sort of (linear) order
  - continuity regarding small state changes

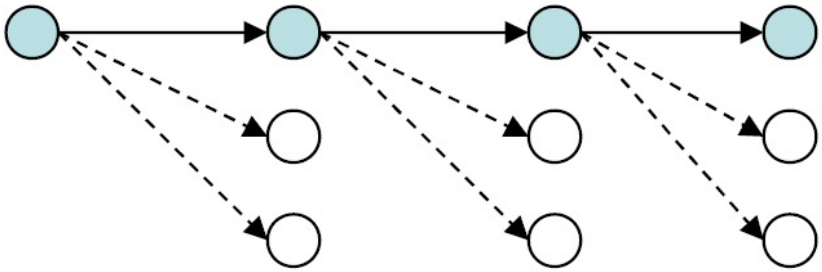
Hill-climbing difficulties

these difficulties apply to all local search algorithms, and usually become much worse as the search space becomes higher dimensional

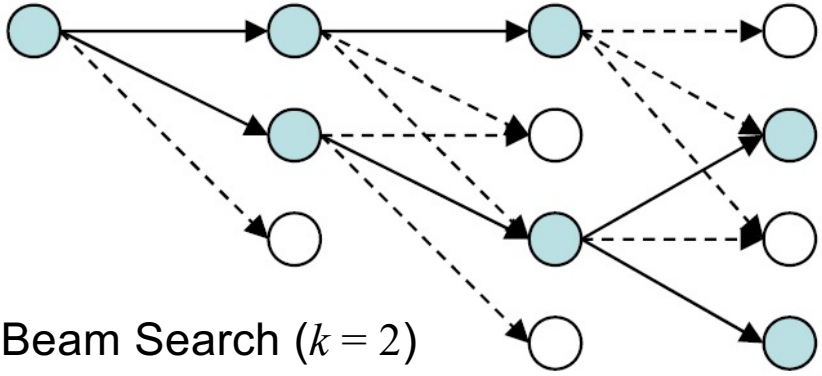


Beam Search

- Keep track of k states rather than just one
  - k is called the beam size
- Algorithm
  - Start with k randomly generated states
  - At each iteration, all the successors of all k states are generated
  - select the k best successors from the complete list and repeat



Hill-Climbing Search



Beam Search ($k = 2$)

Beam Search

- Keep track of k states rather than just one
  - k is called the beam size
- Algorithm
  - Start with k randomly generated states
  - At each iteration, all the successors of all k states are generated
  - select the k best successors from the complete list and repeat.
- Implementation

  Can be implemented similar to the Tree-Search algorithm:
  - sort the queue by the heuristic function h (as in greedy search)
  - but limit the size of the queue to k
  - and expand all nodes in queue simultaneously

Beam Search

- Keep track of k states rather than just one
  - k is called the beam size
- Note
  - Beam search is different from k parallel hill-climbing searches!
  - Information from different beams is combined
- Effectiveness
  - suffers from lack of diversity of the k states
    - e.g., if one state has better successors than all other states
    - thus it is often no more effective than hill-climbing

- Stochastic Beam Search
  - chooses k successors at random
  - better nodes have a higher probability of being selected

Simulated Annealing Search

Idea:
Use conventional hill-climbing style techniques, but occasionally take a step in a direction other than that in which there is improvement (downhill moves; away from solution).

As time passes, the probability that a down-hill step is taken is gradually reduced and the size of any down-hill step taken is decreased.

Simulated Annealing Search

- combination of hill-climbing and random walk
  - escape local maxima by allowing some "bad" moves
  - but gradually decrease their frequency (the *temperature*)

- Effectiveness:
  - it can be proven that if the temperature is lowered slowly enough, the probability of converging to a global optimum approaches 1
  - Unfortunately this can take a VERY VERY long time

  - Note: in any finite search space, random guessing also will find a global optimum with probability approaching 1
  - So, ultimately this is a very weak claim

  - Widely used in VLSI layout, airline scheduling, etc

Simulated Annealing Search

Note:

- Annealing *in metallurgy and materials science, is a heat treatment wherein the microstructure of a material is altered, causing changes in its properties such as strength and hardness. It is a process that produces equilibrium conditions by heating and maintaining at a suitable temperature, and then cooling very slowly.*

Simulated Annealing Search

- combination of hill-climbing and random walk

```
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 ← 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
      ΔE ← VALUE[next] – VALUE[current]
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(Δ E/T)
```

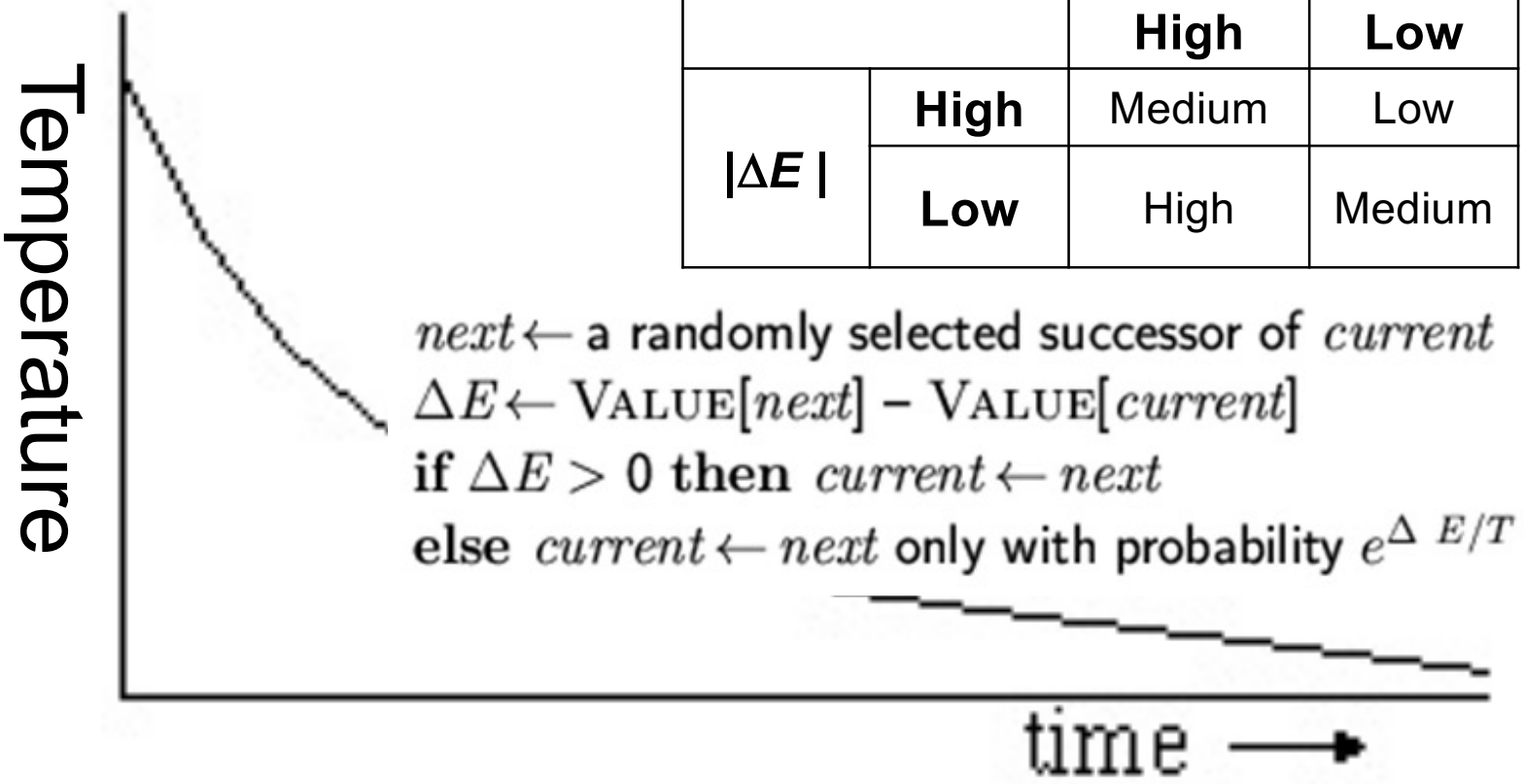
Typical annealing schedule

- Usually use a decaying exponential
- Axis values scaled to fit problem characteristics

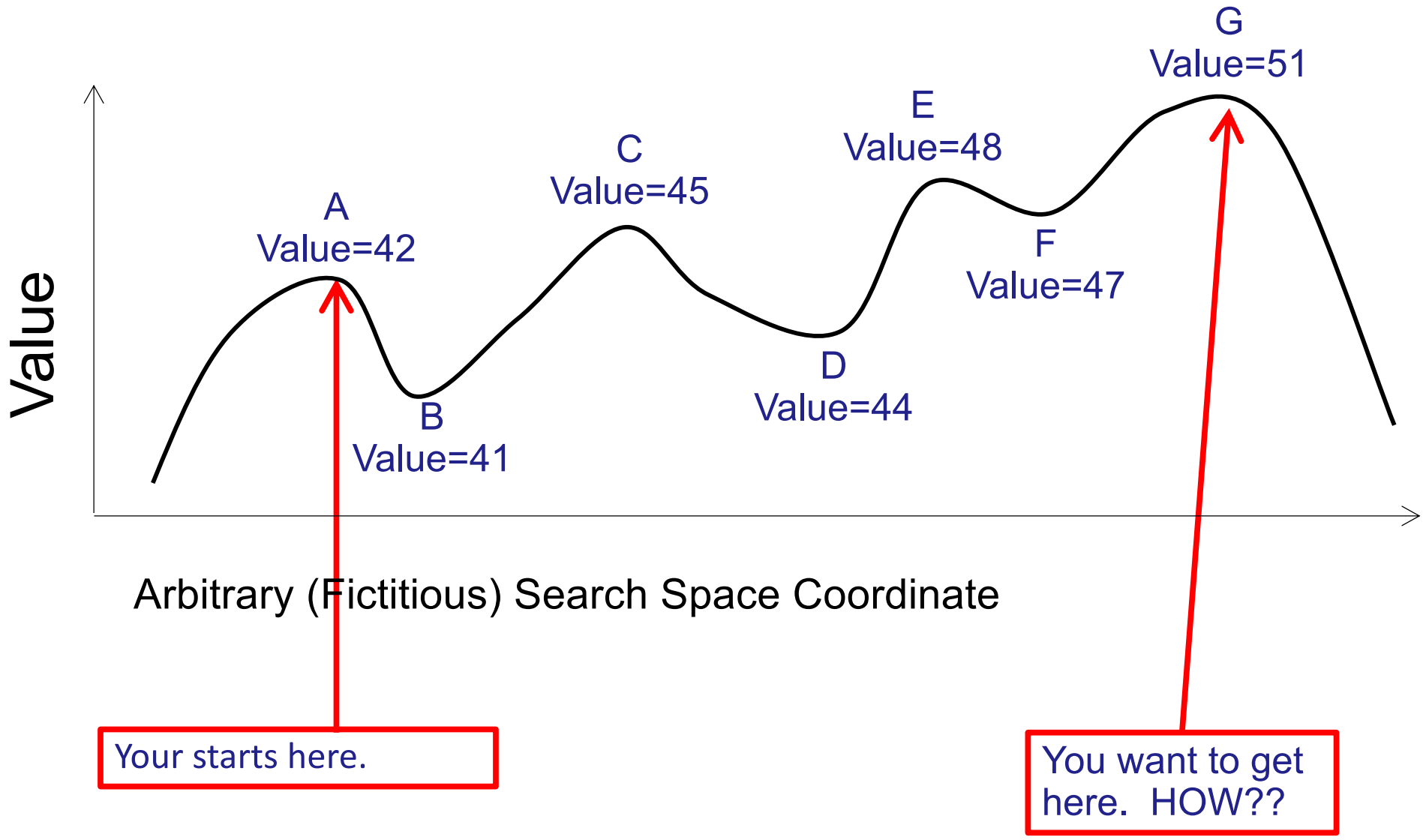
Pr(accept worse successor)

- Decreases as temperature T decreases (accept bad moves early on)
- Increases as |d E| decreases (accept not “much” worse)
- Sometimes, step size also decreases with T

$e^{\Delta E / T}$		Temperature T	
		High	**Low**
$\lvert\Delta E\rvert$	**High**	Medium	Low
	Low	High	Medium

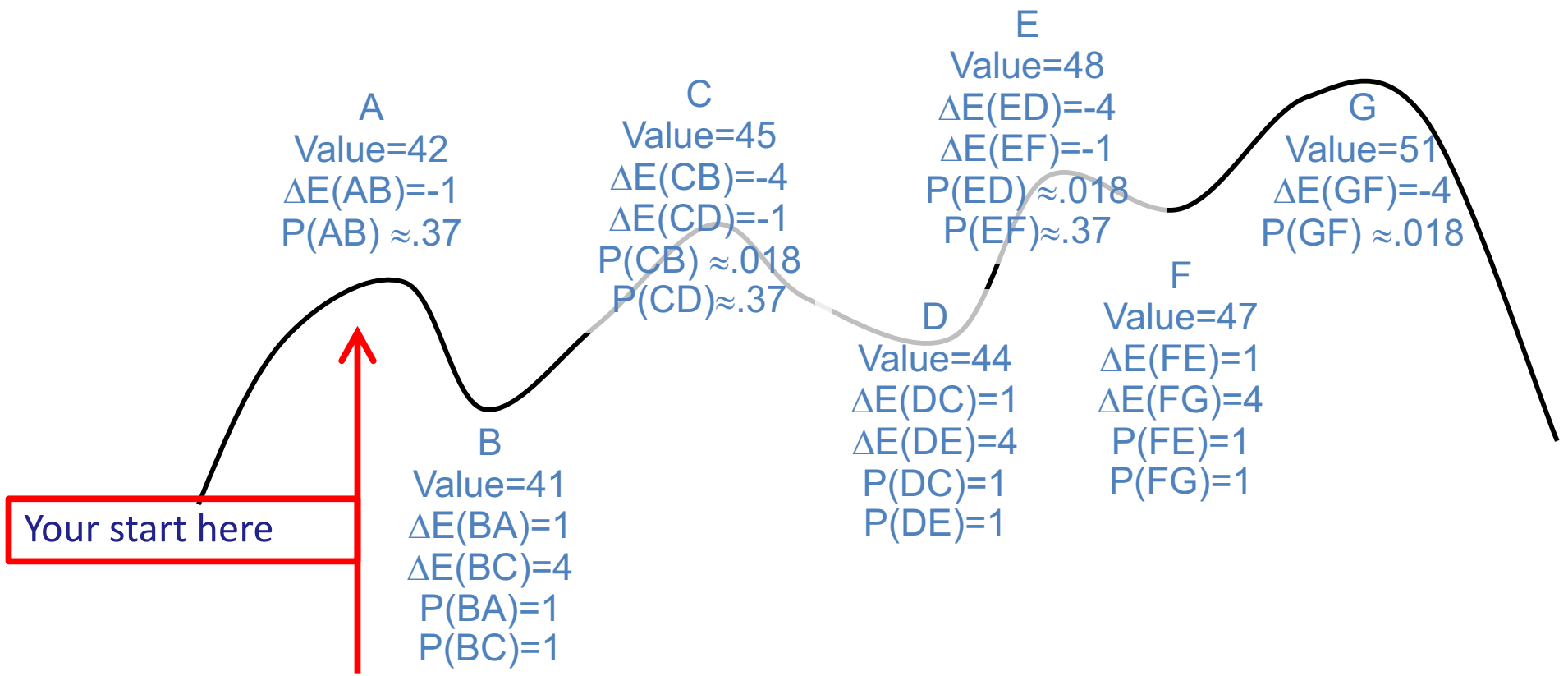


Goal: “ratchet up” a jagged slope



This is an illustrative *cartoon*…

Goal: "ratchet up" a jagged slope



x	-1	-4
e^x	≈.37	≈.018

From A you will accept a move to B with P(AB) ≈.37.
From B you are equally likely to go to A or to C.
From C you are ≈20X more likely to go to D than to B.
From D you are equally likely to go to C or to E.
From E you are ≈20X more likely to go to F than to D.
From F you are equally likely to go to E or to G.
Remember best point you ever found (G or neighbor?).

This is an illustrative *cartoon*…

Properties of simulated annealing

- One can prove:
  - If *T* decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1
  - Unfortunately this can take a VERY VERY long time
  - Note: in any finite search space, random guessing also will find a global optimum with probability approaching 1
  - So, ultimately this is a very weak claim
- Often works very well in practice
  - But usually VERY VERY slow
- Widely used in VLSI layout, airline scheduling, etc.

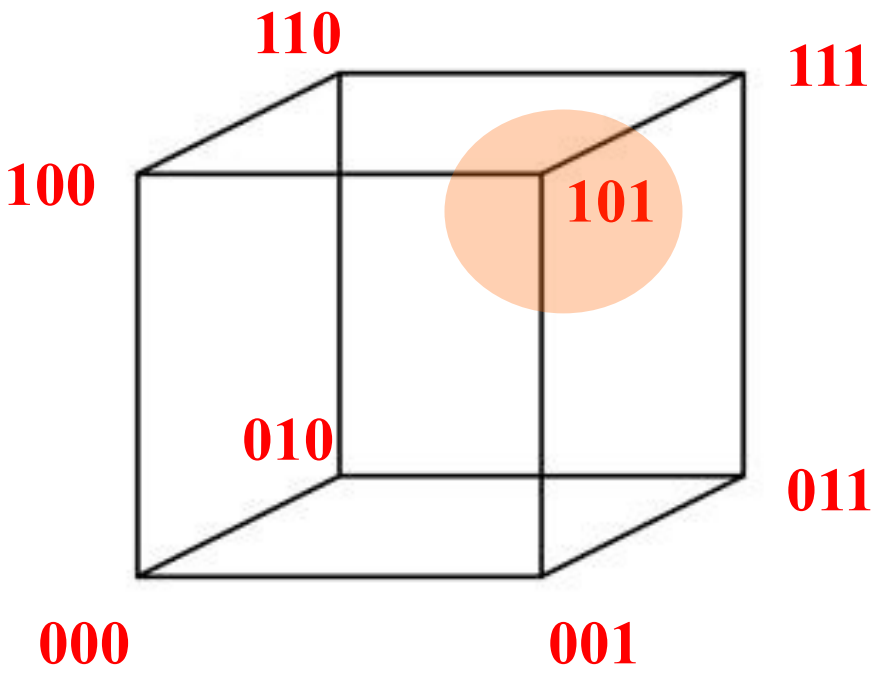
Simulated Annealing (SA)

- Superficially: SA is local search with some noise added. Noise starts high and is slowly decreased.

- **True story is much more principled:**
  SA is a general sampling strategy to sample from a (combinatorial) space according to a well-defined probability distribution.

- **Sampling strategy models the way physical systems, such as gases, sample from their statistical equilibrium distributions. Order 10^23 particles. Studied in the field of statistical physics.**

Example: 3D Hypercube space



States		Value f(s)
s1	000	2
s2	001	4.25
s3	010	4
s4	011	3
s5	100	2.5
s6	101	4.5
s7	110	3
s8	111	3.5

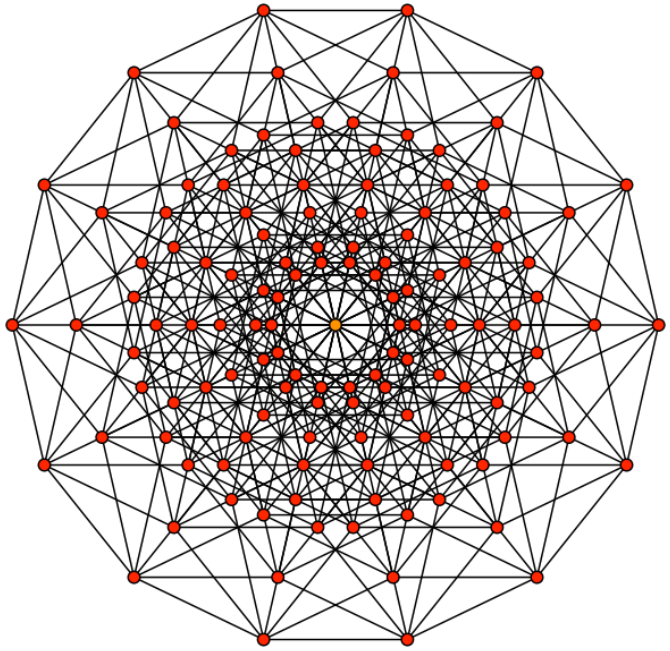
N dimensional “hypercube” space. N =3. 2^3 = 8 states total.

Goal: Optimize f(s), the value function. Maximum value 4.5 in s6.

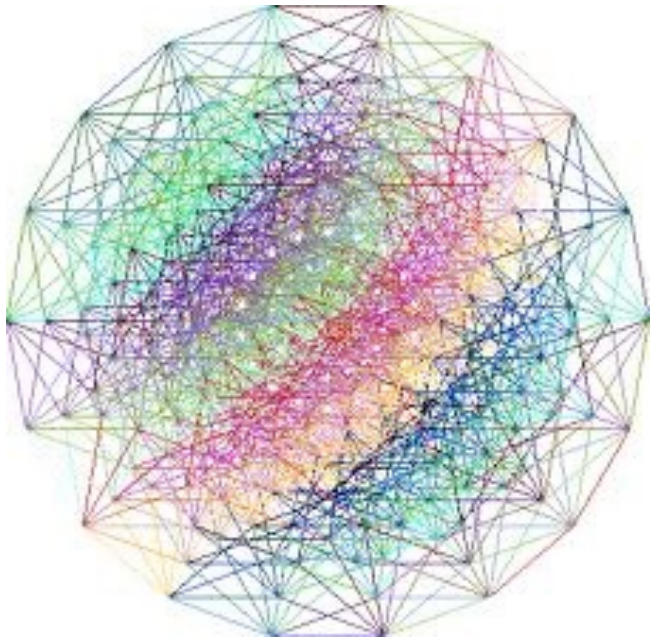
Use local search: Each state / node has N = 3 neighbors (out of 2^N total).

Of course, real interest in large N…

- Spaces with 2^N states and each state with N neighbors.



7D hypercube; 128 states.
Every node, connected to 7 others.
Max distance between two nodes: 7.



9D hypercube; 512 states.

Practical reasoning problem: N = 1,000,000. 2^N = 10^300,000

SA node sampling strategy

Consider the following "random walker" in hypercube space:

1) **Start at a random node S (the "current node").**

   (How do we generate such a node?)

2) **Select, at random, one of the N neighbors of S, call it S'**

If (f(S') – f(S)) > 0, move to S', i.e. set S := S'

(i.e., jump to node with better value)

else with prob. e^(f(S')-f(S))/T move to S', i.e., set S := S'

4) **Go back to 2)**

Note: Walker keeps going and going. Does not get stuck in any one node.

Prob(s) = e^(f(s))/T / Z

For our example space

States		Value f(s)	T=1.0	Prob(s)	T=0.5	Prob(s)
s1	000	2	7.4	0.02	55	0.003
s2	001	4.25	70.1	0.23	4915	0.27
s3	010	4	54.6	0.18	2981	0.17
s4	011	3	20.1	0.07	403	0.02
s5	100	2.5	12.2	0.04	148	0.008
s6	101	4.5	90.0	0.29	8103	0.45
s7	110	3	20.1	0.07	403	0.02
s8	111	3.5	33.1	0.11	1097	0.06
			sum Z = 307.9		sum Z = 18,105	

So, at T = 1.0, walker will spend roughly 29% of its time in the best state.

T = 0.5, roughly 45% of its time in the best state.

Prob(s) = e^(f(s))/T / Z

For our example space

States		Value f(s)	T=1.0	Prob(s)	T=0.25	Prob(s)
s1	000	2	7.4	0.02	2981	0.000
s2	001	4.25	70.1	0.23	24,154,952	0.24
s3	010	4	54.6	0.18	8,886,111	0.09
s4	011	3	20.1	0.07	162,755	0.001
s5	100	2.5	12.2	0.04	22,026	0.008
s6	101	4.5	90.0	0.29	65,659,969	0.65
s7	110	3	20.1	0.07	162,755	0.001
s8	111	3.5	33.1	0.11	1,202,604	0.06
			sum Z = 307.9		sum Z = 100,254,153	

So, at T = 1.0, walker will spend roughly 29% of its time in the best state.

T = 0.5, roughly 45% of its time in the best state.

T = 0.25, roughly 65% of its time in the best state.
And, remaining time mostly in s2 (2nd best)!

So, when T gets lowered, the probability distribution starts to "concentrate" on the maximum (and close to maximum) value states.

The lower T, the stronger the effect!

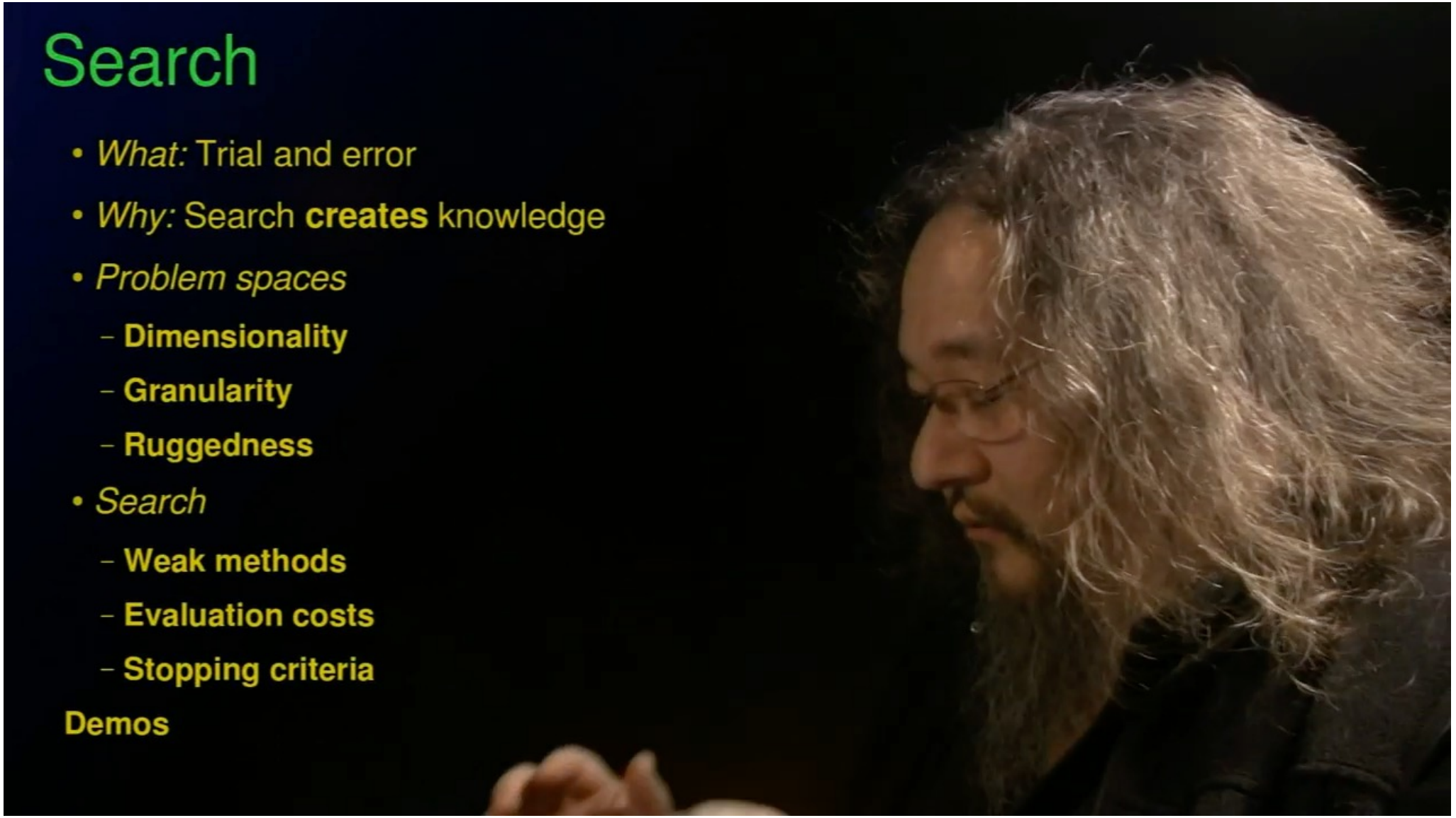
At low T, we can just output the current state. It will quite likely be a maximum value (or close to it) state.

In practice: Keep track of best state seen during the SA search.

SA is an example of Markov Chain Monte Carlo (MCMC) sampling.

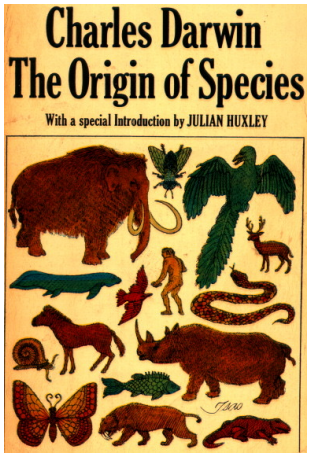
It's very general technique to sample from complex probability distributions by making local moves only. For optimization, we chose a clever probability distribution that *concentrates on the optimum states for low T.* (Kirkpatrick et al. 1984)

Dave Ackley, New Mexico CS for all, UNM CS
https://www.youtube.com/watch?v=boTeFM-CVFw



Genetic Algorithms

- Another class of iterative improvement algorithms
  - A genetic algorithm maintains a population of candidate solutions for the problem at hand, and makes it evolve by iteratively applying a set of stochastic operators
- Inspired by the biological evolution process
- Uses concepts of "Natural Selection" and "Genetic Inheritance" (Darwin 1859)
- Originally developed by John Holland (1975)



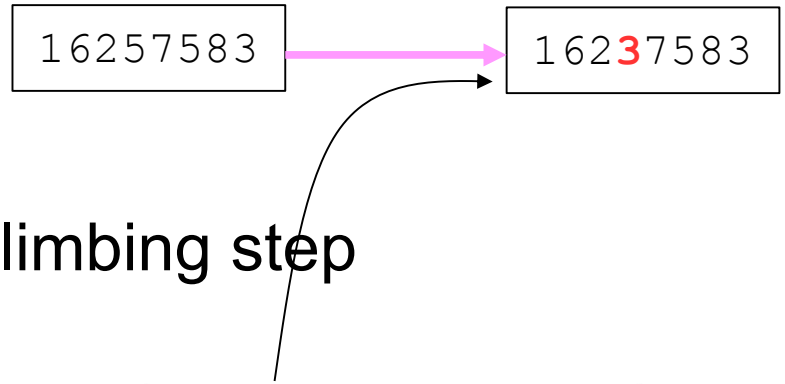
Genetic Algorithms

- The idea is akin to Stochastic Beam Search
  - but we now use „sexual“ reproduction (new nodes have two parents)

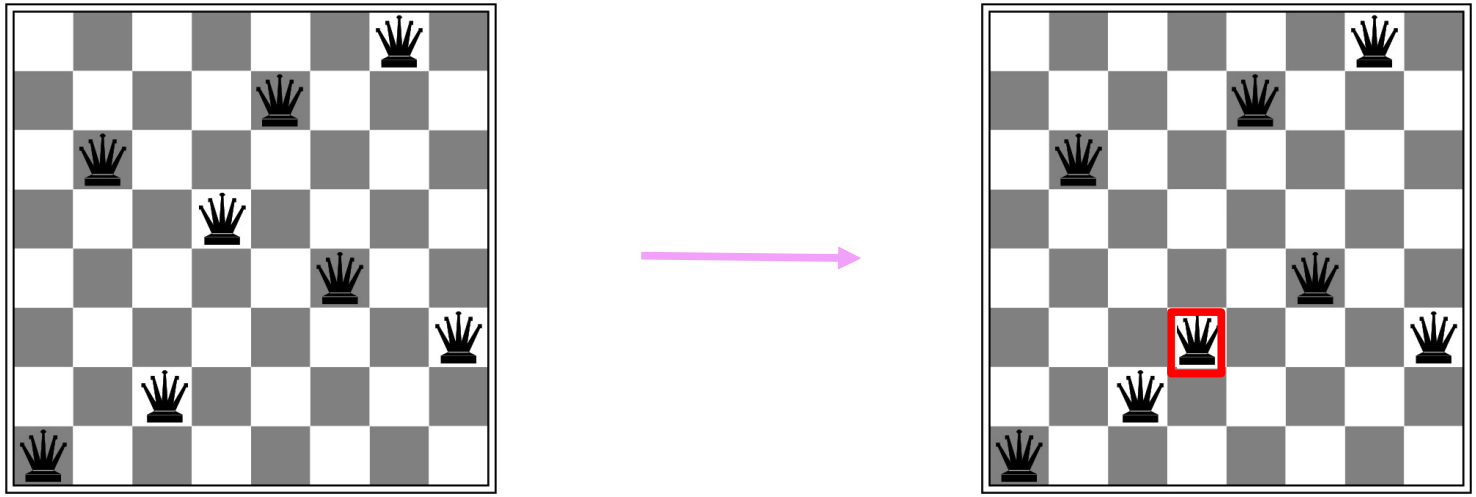
- Basic Algorithm:
  - Start with k randomly generated states (population)
  - A state is represented as a string over a finite alphabet
    - often a string of 0s and 1s
  - Evaluation function (fitness function)
  - Produce the next generation by selection, cross-over, and mutation

Mutation

- Modelled after mutation of DNA
  - take one parent strings
  - modify a random value

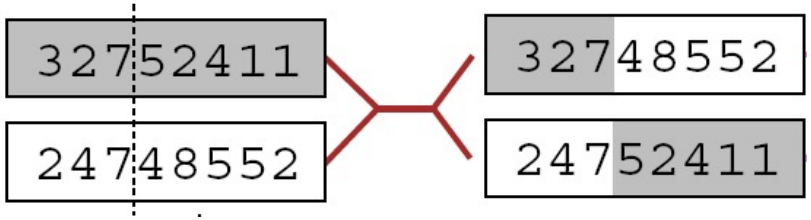


- comparable to a stochastic hill-climbing step

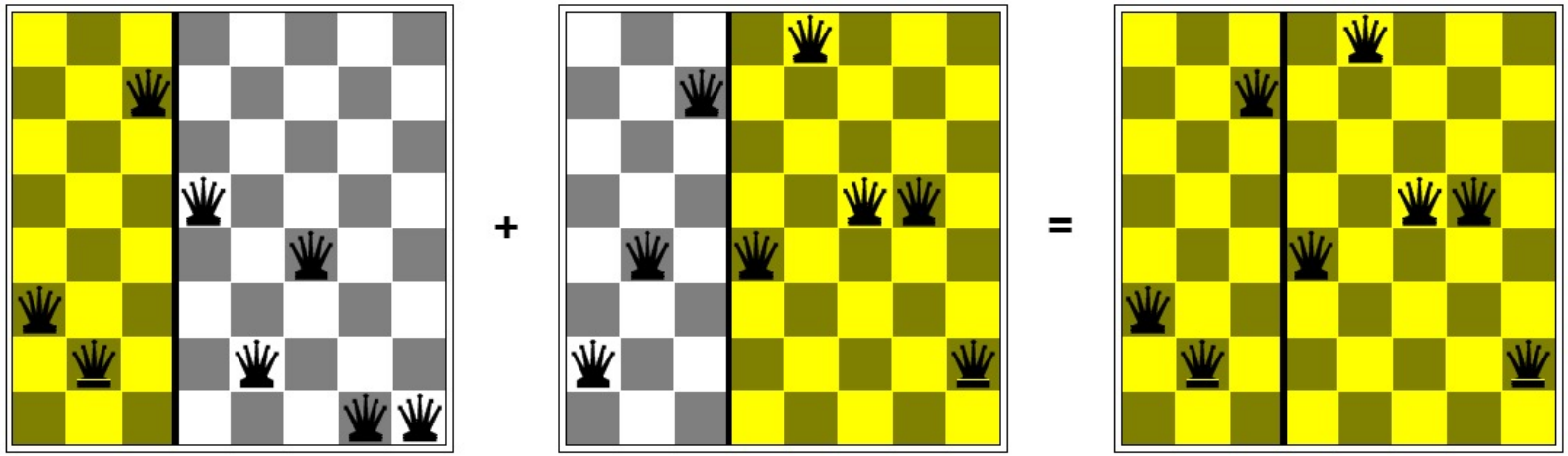


Cross-Over

- Modelled after cross-over of DNA
  - take two parent strings
  - cut them at cross-over point
  - recombine the pieces

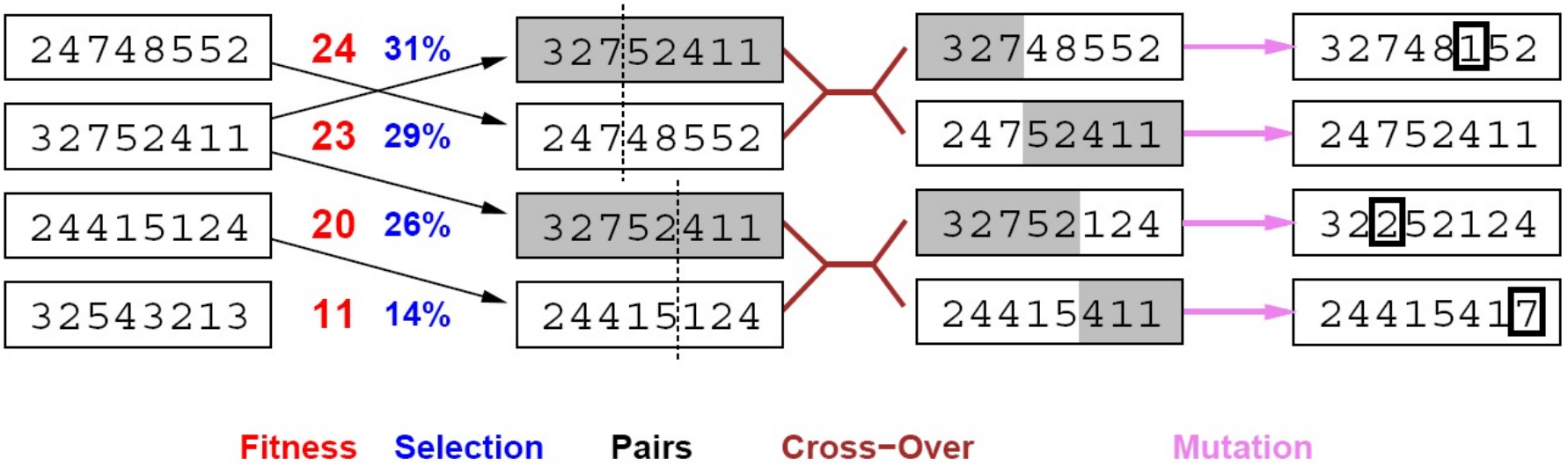


- it is helpful if the substrings are meaningful subconcepts



Genetic Algorithms

- Same idea as in Stochastic Beam Search
  - but uses „sexual“ reproduction (new nodes have two parents)
- Basic Algorithm:
  - Start with k randomly generated states (population)
  - A state is represented as a string over a finite alphabet
    - often a string of 0s and 1s
  - Evaluation function (fitness function)
  - Produce the next generation by selection, cross-over, and mutation

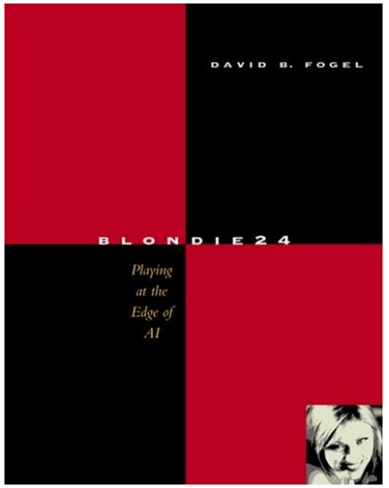


Genetic Algorithm

function GENETIC_ALGORITHM(*population,* FITNESS-FN) **return** an individual
  input: *population*, a set of individuals
    FITNESS-FN, a function which determines the quality of the individual
  repeat
    new_population ← empty set
    loop for i **from** 1 **to** SIZE(*population*) **do**
      x ← RANDOM_SELECTION(*population*, FITNESS_FN)
      y ← RANDOM_SELECTION(*population*, FITNESS_FN)
      child ← REPRODUCE(*x*,*y*)
      if (small random probability) **then** *child* ← MUTATE(*child*)
      add *child* to *new_population*
    population ← *new_population*
  until some individual is fit enough or enough time has elapsed
  return the best individual in *population*, according to FITNESS_FN

Genetic Algorithms

- Evaluation
  - attractive and popular
    - easy to implement general optimization algorithm
    - easy to explain to laymen (boss)
  - perform well
    - unclear under which conditions they work well
    - other randomized algorithms perform equally well (or better)
- Numerous applications
  - optimization problems
    - circuit layout
    - job-shop scheduling
  - game playing
    - checkers program Blondie24 (David Fogel)
      - nice and easy read, but shooting a bit over target in its claims...



Genetic Programming

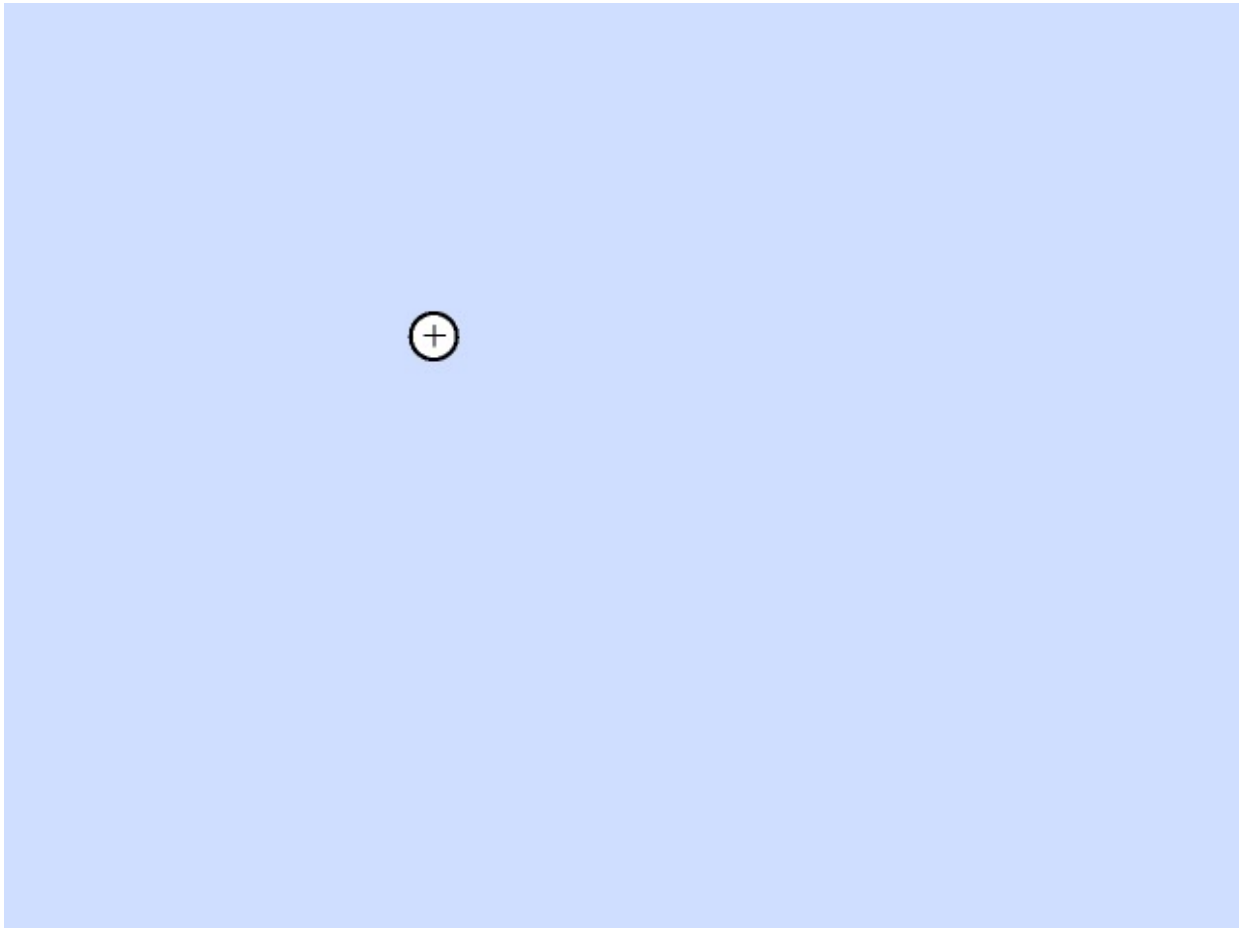
- popularized by John R. Koza

> *Genetic programming is an automated method for creating a working computer program from a high-level problem statement of a problem. It starts from a high-level statement of "what needs to be done" and automatically creates a computer program to solve the problem.*

- applies Genetic Algorithms to program trees
  - Mutation and Cross-over adapated to tree structures
  - special operations like
    - inventing/deleting a subroutine
    - deleting/adding an argument,
    - etc.

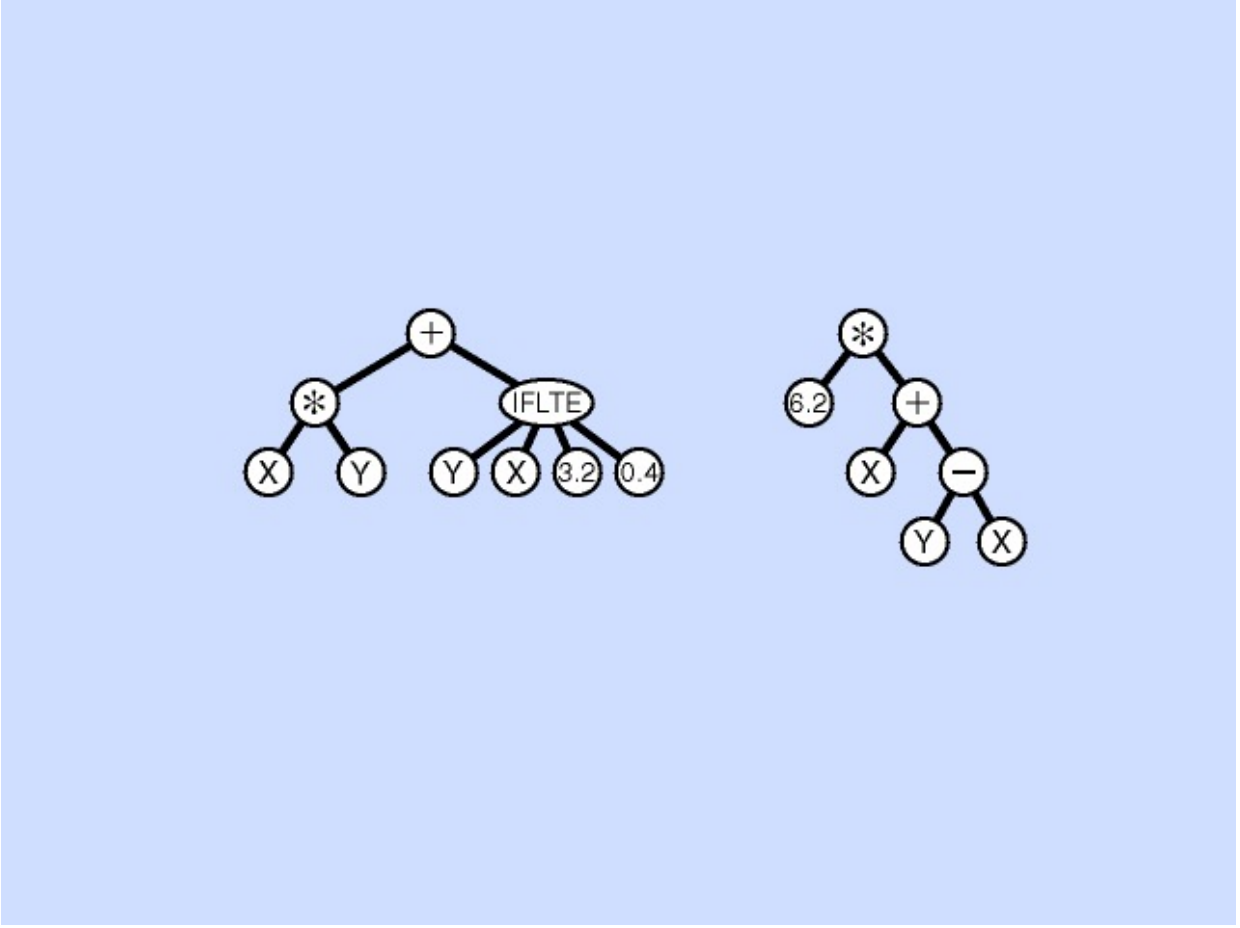
- Several successful applications
  - Annual awards for performance competitive to humans http://www.genetic-programming.com/humancompetitive.html
- More information at http://www.genetic-programming.org/

Random Initialization of Population



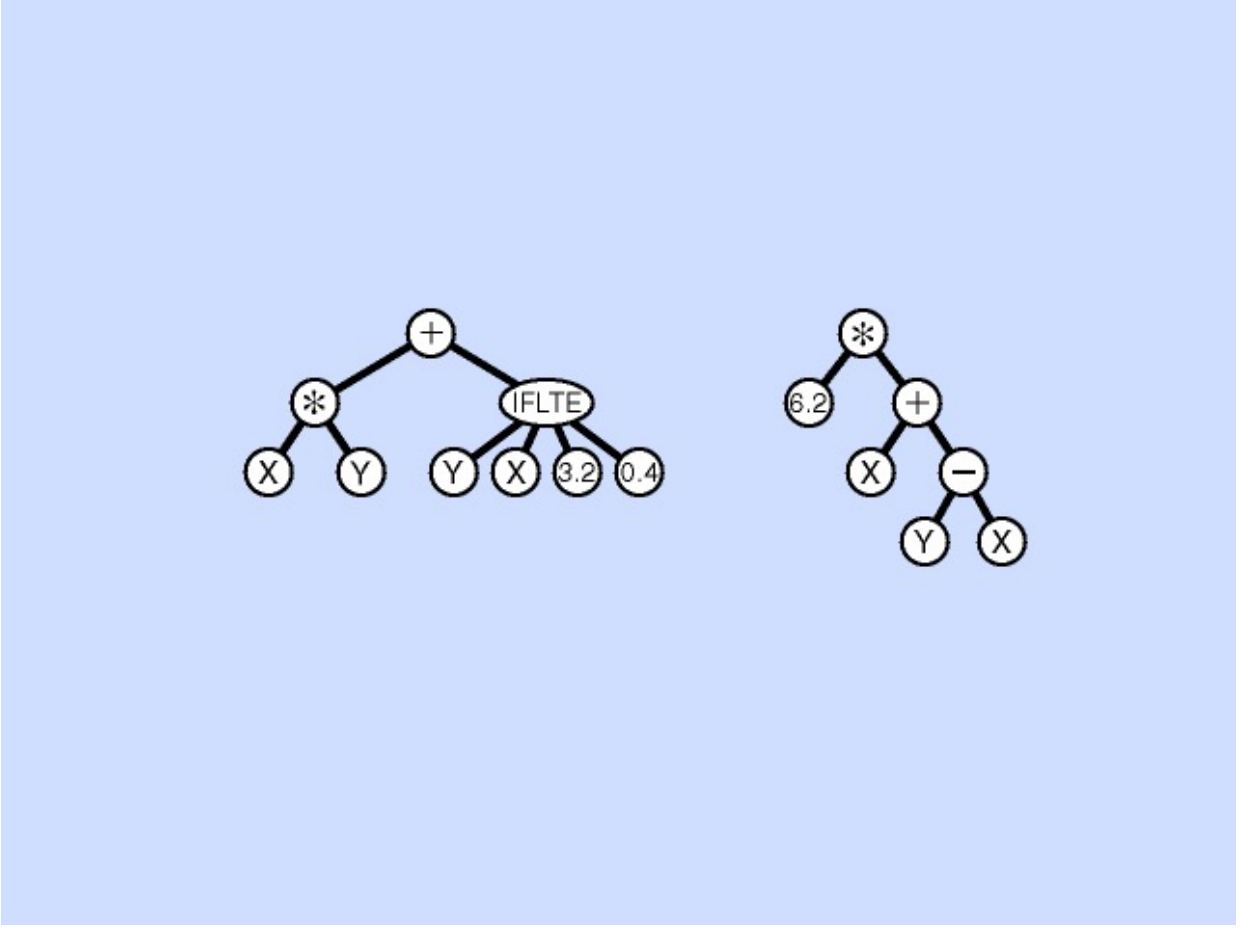
Animated Image taken from http://www.genetic-programming.com/gpanimatedtutorial.html

Mutation



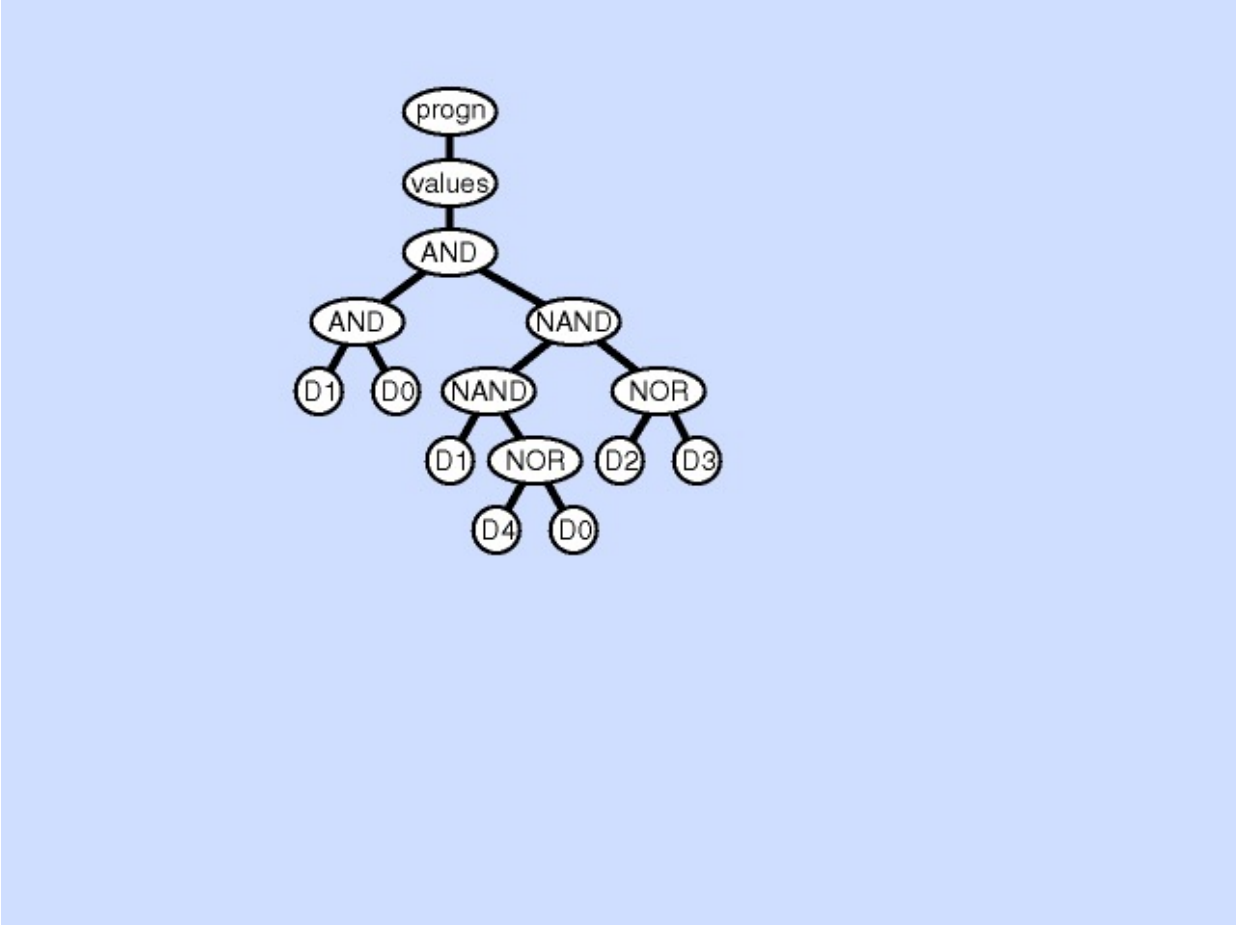
Animated Image taken from http://www.genetic-programming.com/gpanimatedtutorial.html

Cross-Over



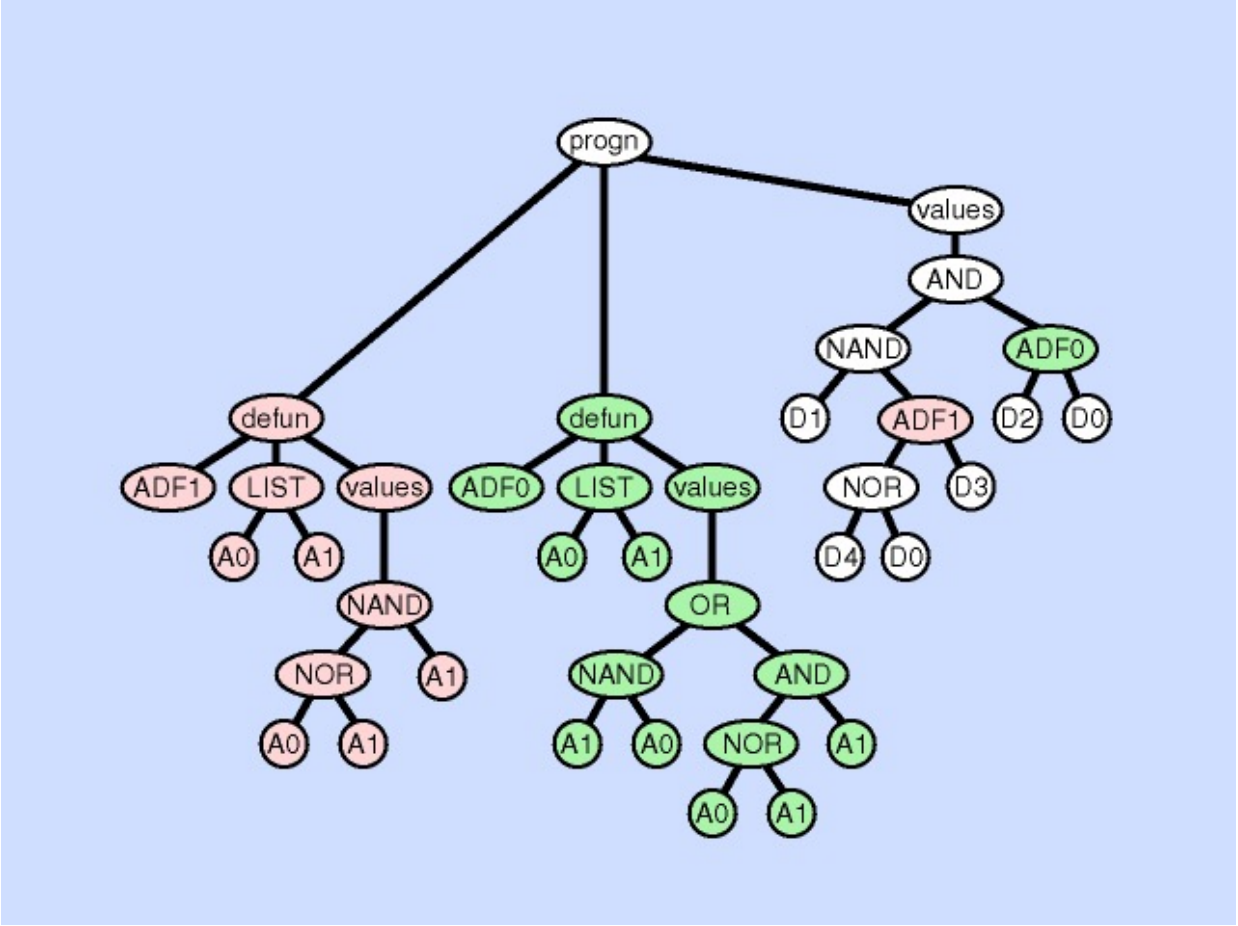
Animated Image taken from http://www.genetic-programming.com/gpanimatedtutorial.html

Create a Subroutine



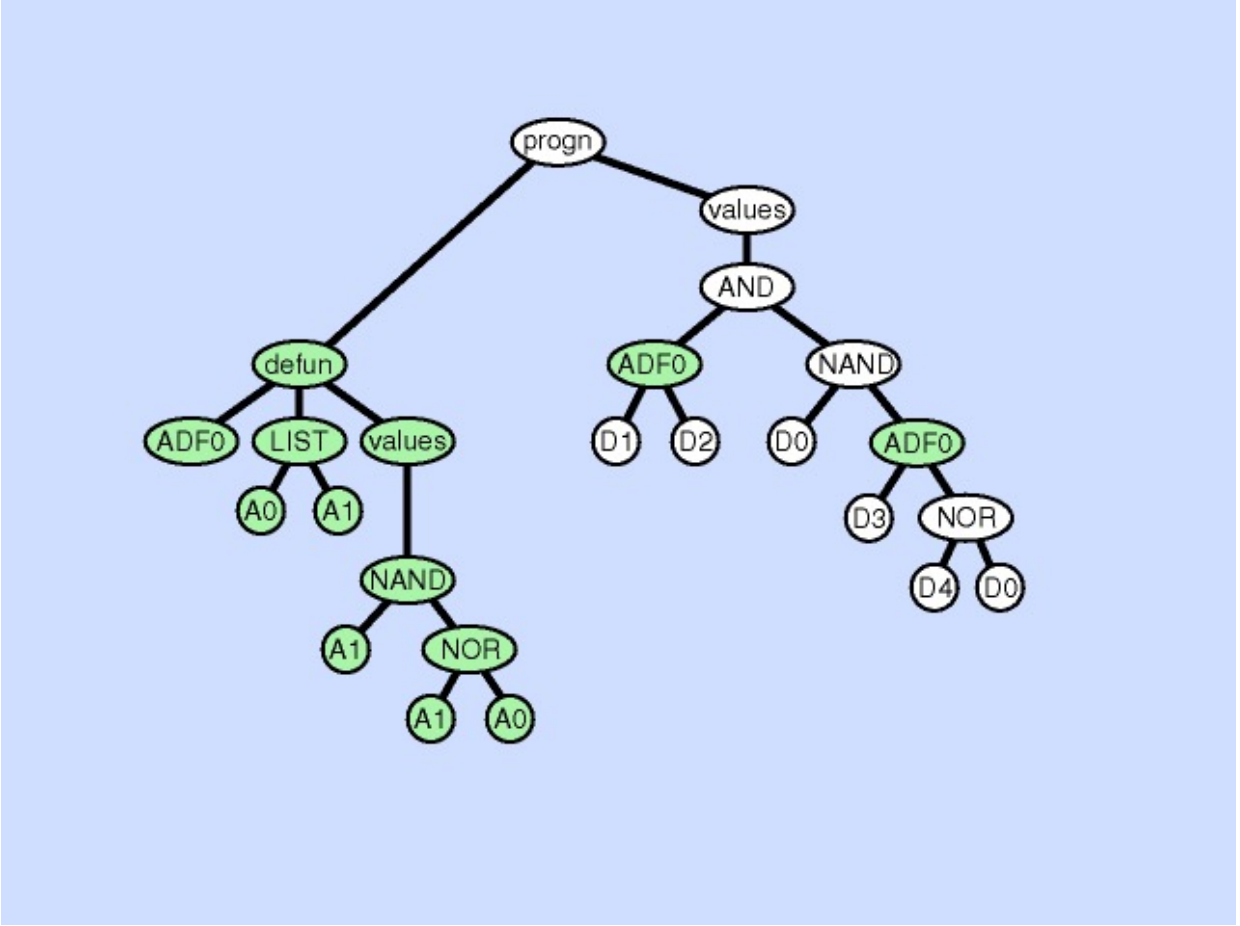
Animated Image taken from http://www.genetic-programming.com/gpanimatedtutorial.html

Delete a Subroutine



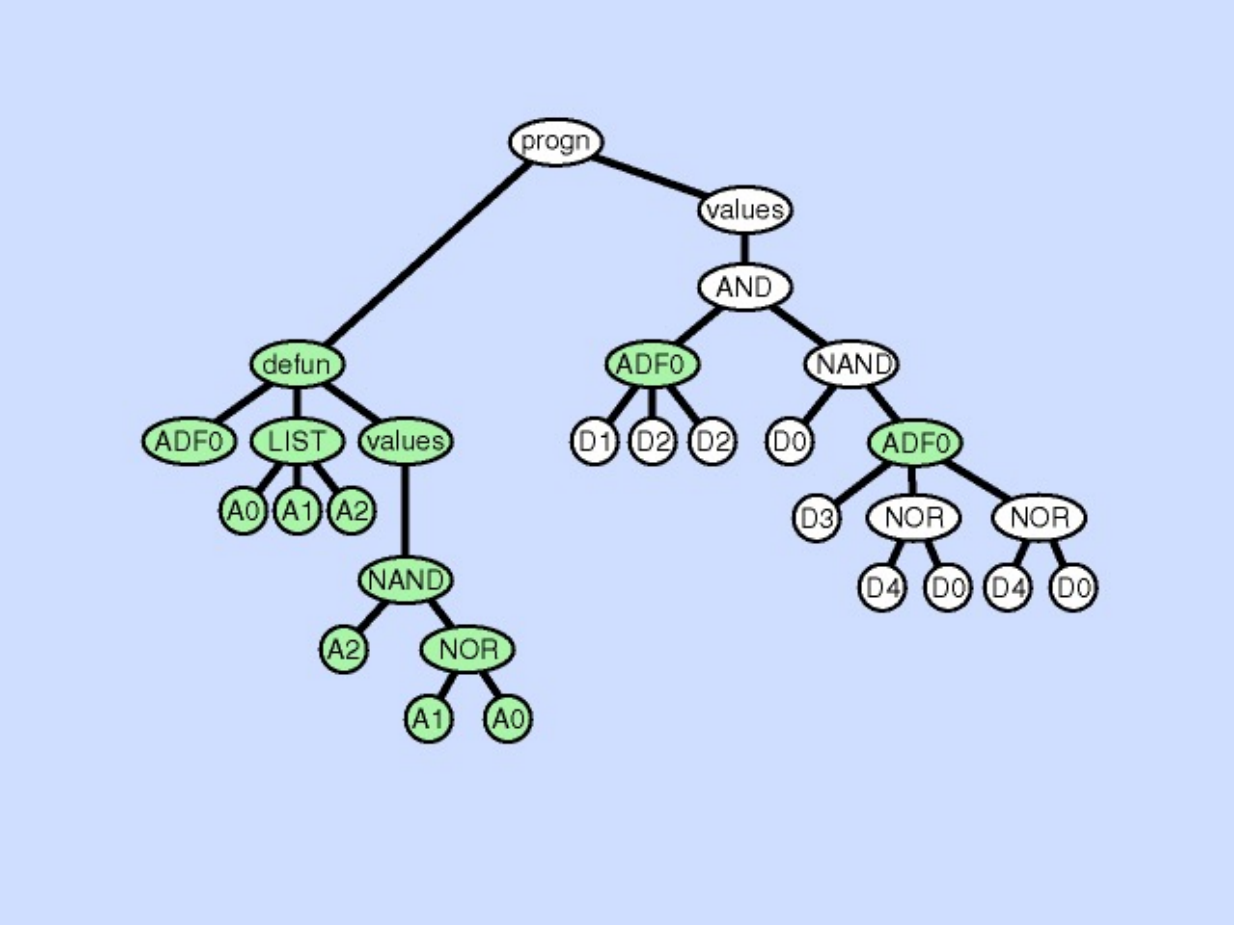
Animated Image taken from http://www.genetic-programming.com/gpanimatedtutorial.html

Duplicate an Argument



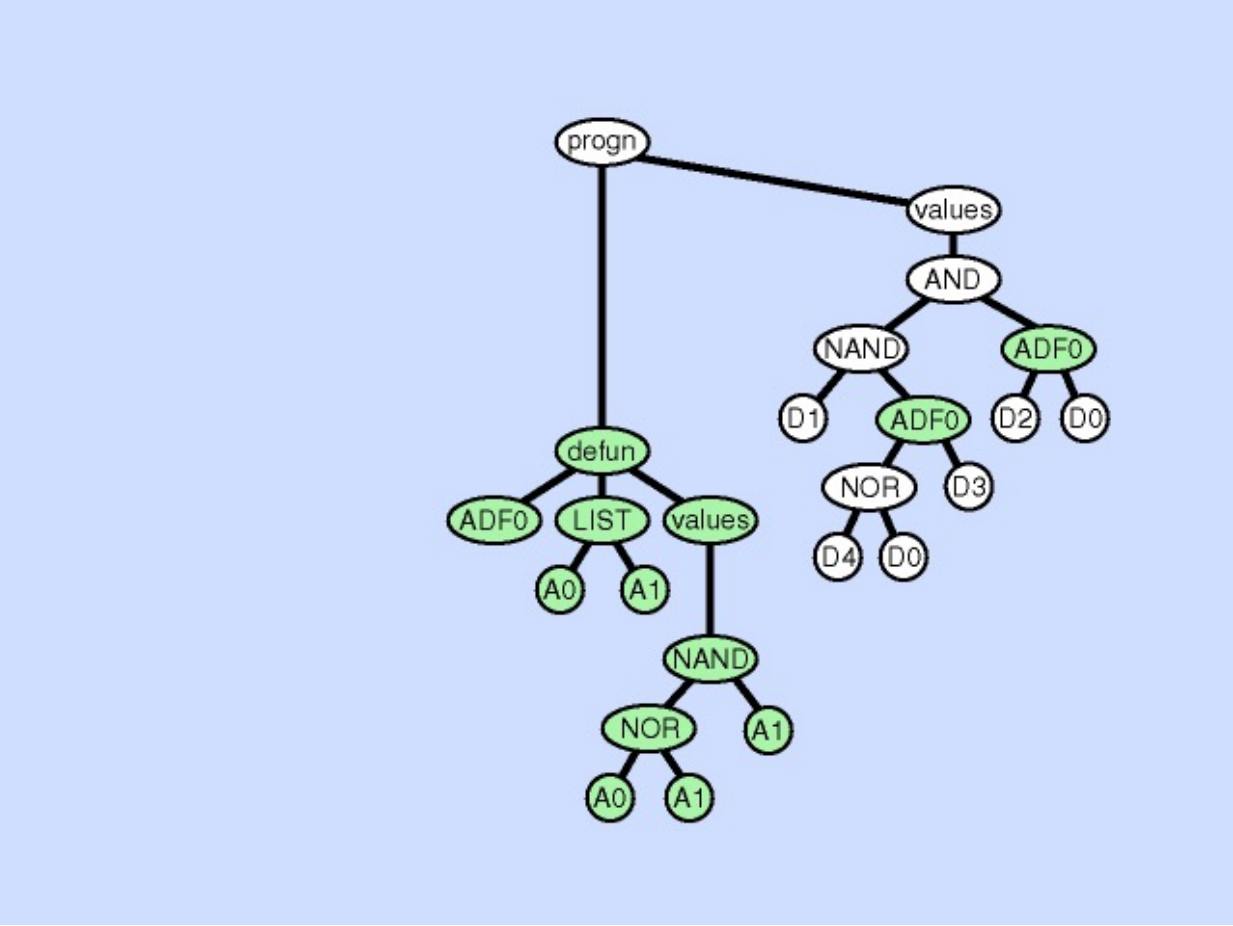
Animated Image taken from http://www.genetic-programming.com/gpanimatedtutorial.html

Delete an Argument



Animated Image taken from http://www.genetic-programming.com/gpanimatedtutorial.html

Create a Subroutine by Duplication



Animated Image taken from http://www.genetic-programming.com/gpanimatedtutorial.html

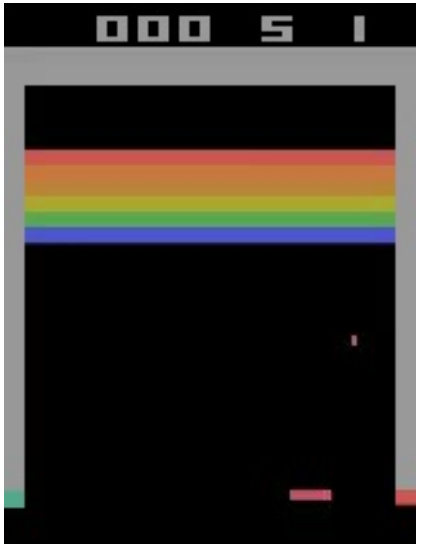
Might be relevant for Deep Reinforcement Learning

Improving Exploration in Evolution Strategies for Deep Reinforcement Learning via a Population of Novelty-Seeking Agents

Edoardo Conti* **Vashisht Madhavan*** **Felipe Petroski Such**
Joel Lehman **Kenneth O. Stanley** **Jeff Clune**
Uber AI Labs

Abstract

Evolution strategies (ES) are a family of black-box optimization algorithms able to train deep neural networks roughly as well as Q-learning and policy gradient methods on challenging deep reinforcement learning (RL) problems, but are much faster (e.g. hours vs. days) because they parallelize better. However, many RL problems require directed exploration because they have reward functions that are sparse or deceptive (i.e. contain local optima), and it is unknown how to encourage such exploration with ES. Here we show that algorithms that have been invented to promote directed exploration in small-scale evolved neural networks via populations of exploring agents, specifically novelty search (NS) and quality diversity (QD) algorithms, can be hybridized with ES to improve its performance on sparse or deceptive deep RL tasks, while retaining scalability. Our experiments confirm that the resultant new algorithms, NS-ES and two QD algorithms, NSR-ES and NSRA-ES, avoid local optima encountered by ES to achieve higher performance on Atari and simulated robots learning to walk around a deceptive trap. This paper



Edoardo Conti, Vashisht Madhavan, Felipe Petroski Such, Joel Lehman, Kenneth O. Stanley, Jeff Clune: Improving Exploration in Evolution Strategies for Deep Reinforcement Learning via a Population of Novelty-Seeking Agents. NeurIPS 2018: 5032-5043