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

Local Search Algorithms

- 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

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

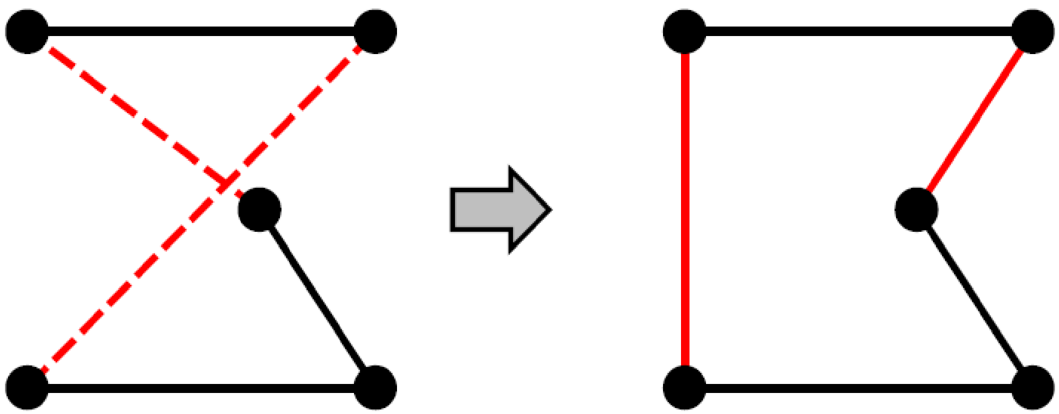
Optimization Problems

- 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

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

Example: Travelling Salesman Problem

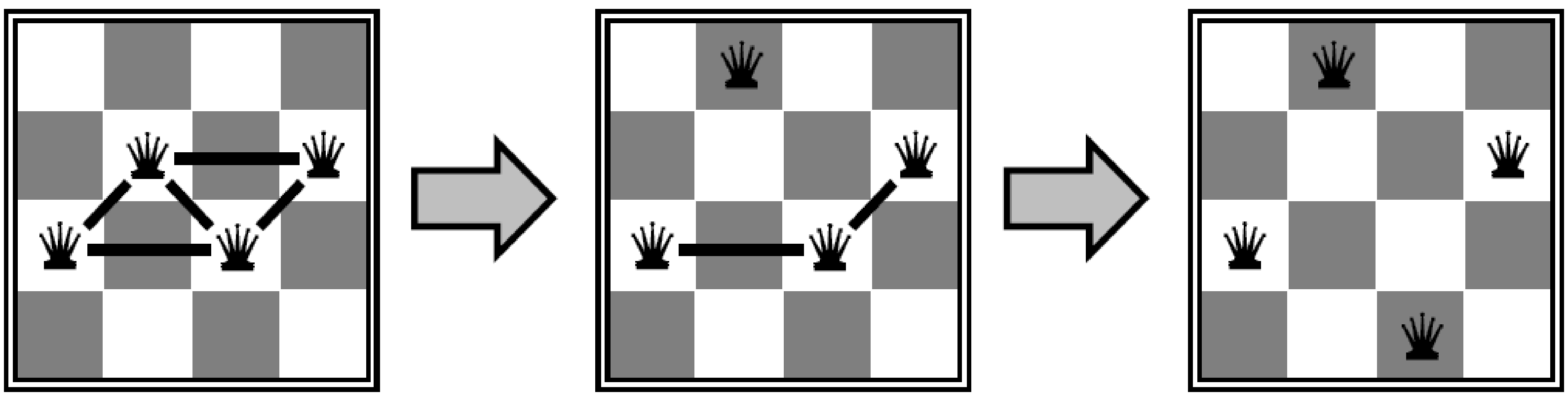
- Basic Idea:
  - Start with a complete tour
  - perform pairwise exchanges



  - 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 h = 2 h = 0

  - 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)

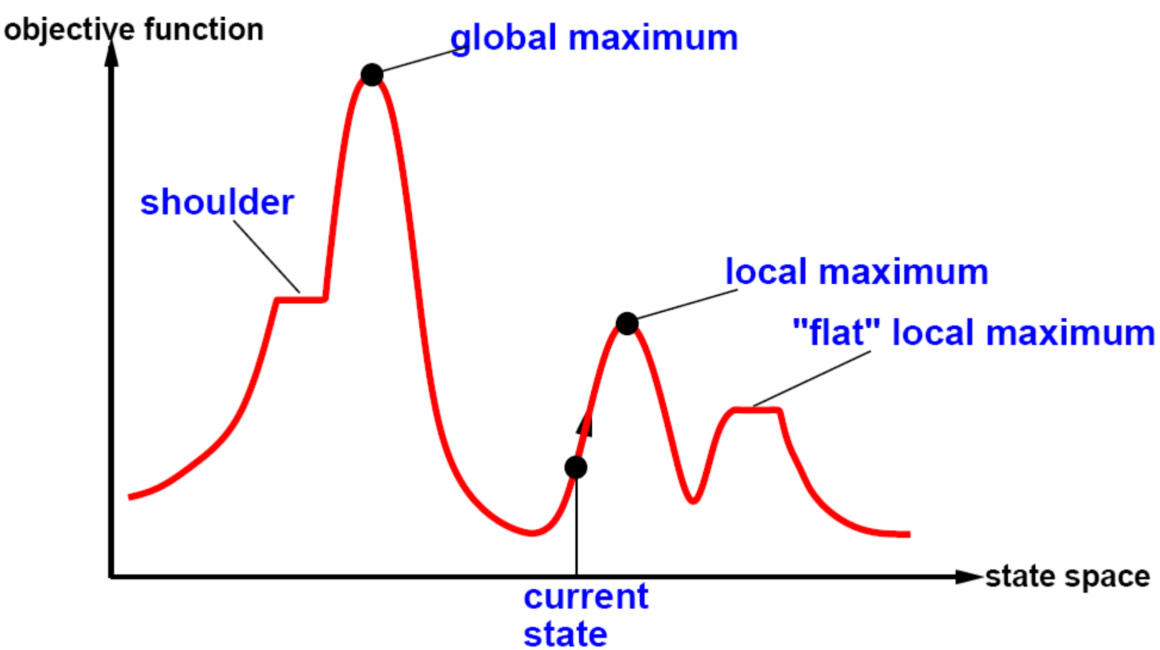
- Algorithm:
  - expand the current state (generate all neighbors)
  - move to the one with the highest evaluation
  - until the evaluation goes down
- Main Problem: Local Optima
  - the algorithm will stop as soon as is at the top of a hill
  - but it is actually looking for a mountain peak

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

- Other problems:
  - ridges
  - plateaux
  - shoulders

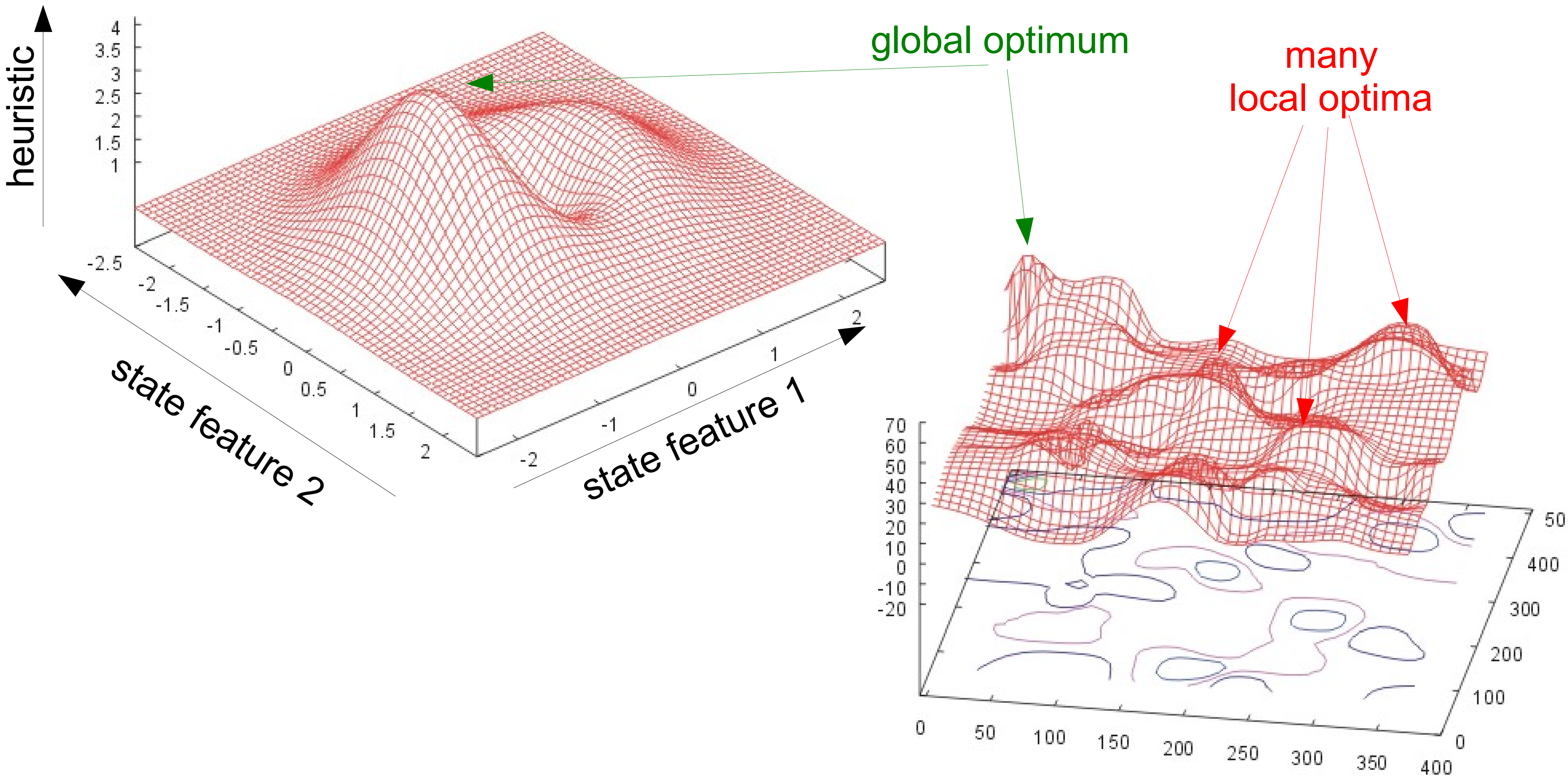
State Space Landscape

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



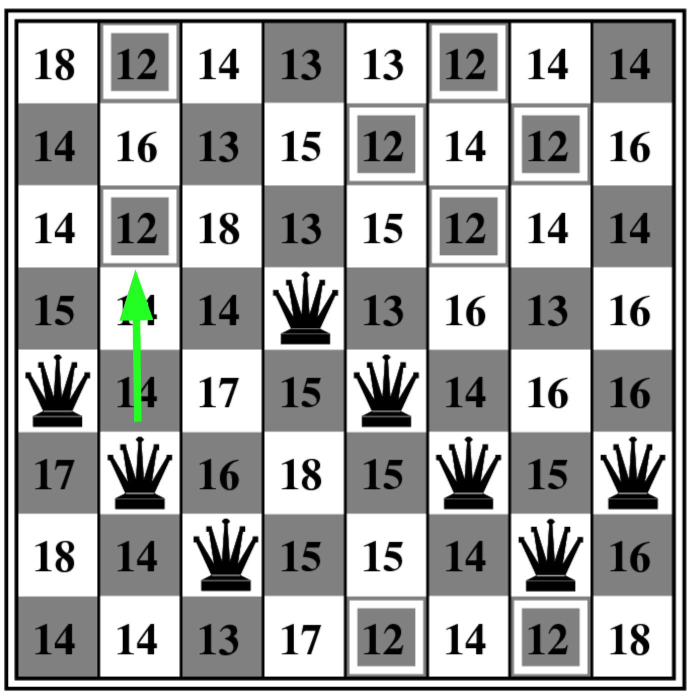
Multi-Dimensional State-Landscape

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



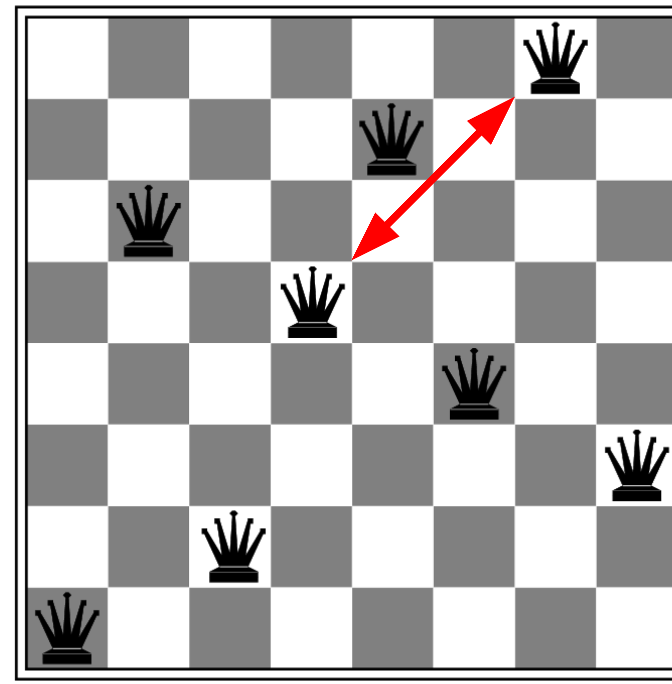
Example: 8-Queens Problem

- Heuristic h:
  - number of pairs of queens that attach each other
- 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

Randomized Hill-Climbing Variants

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

  $\rightarrow$ 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

  $\rightarrow$ 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

Simulated Annealing Search

- combination of hill-climbing and random walk
- Idea:
  - 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
  - Widely used in VLSI layout, airline scheduling, etc

- 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 $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E/T}$

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
  - If any one is a goal state, stop; else 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

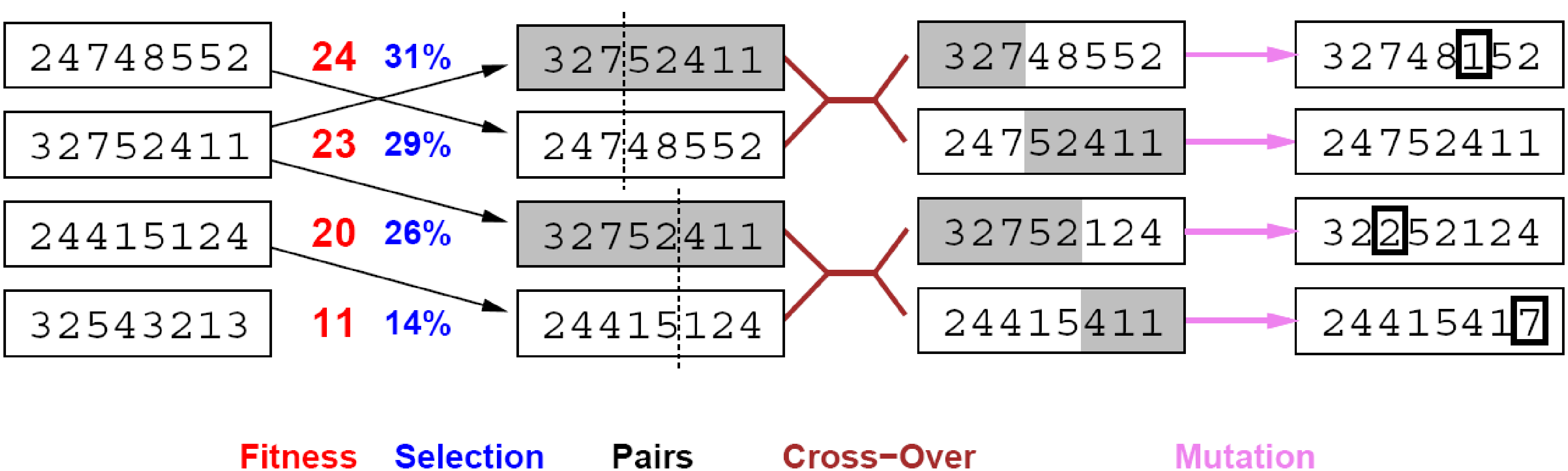
Beam Search

- 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

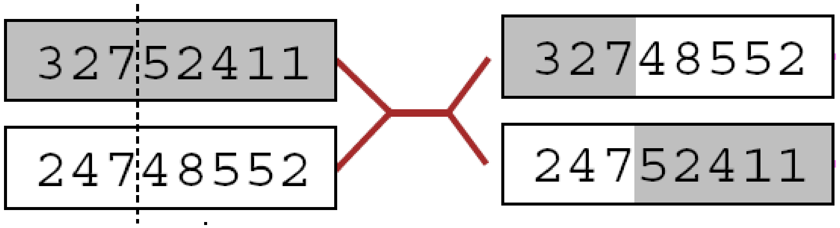
Genetic Algorithms

- 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

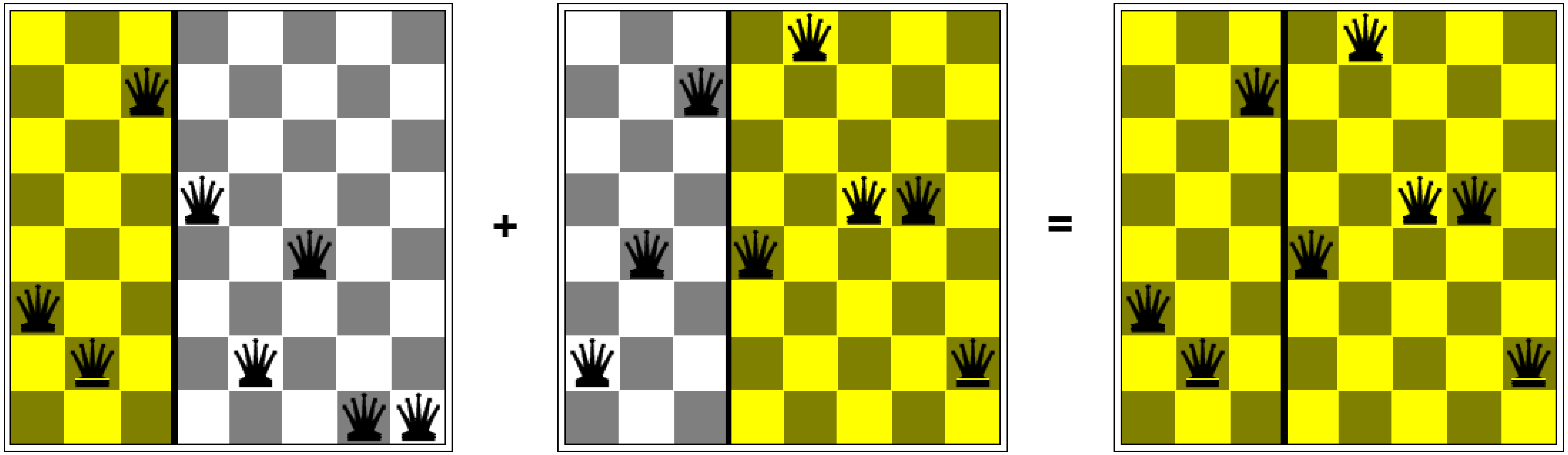


Cross-Over

- 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 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)

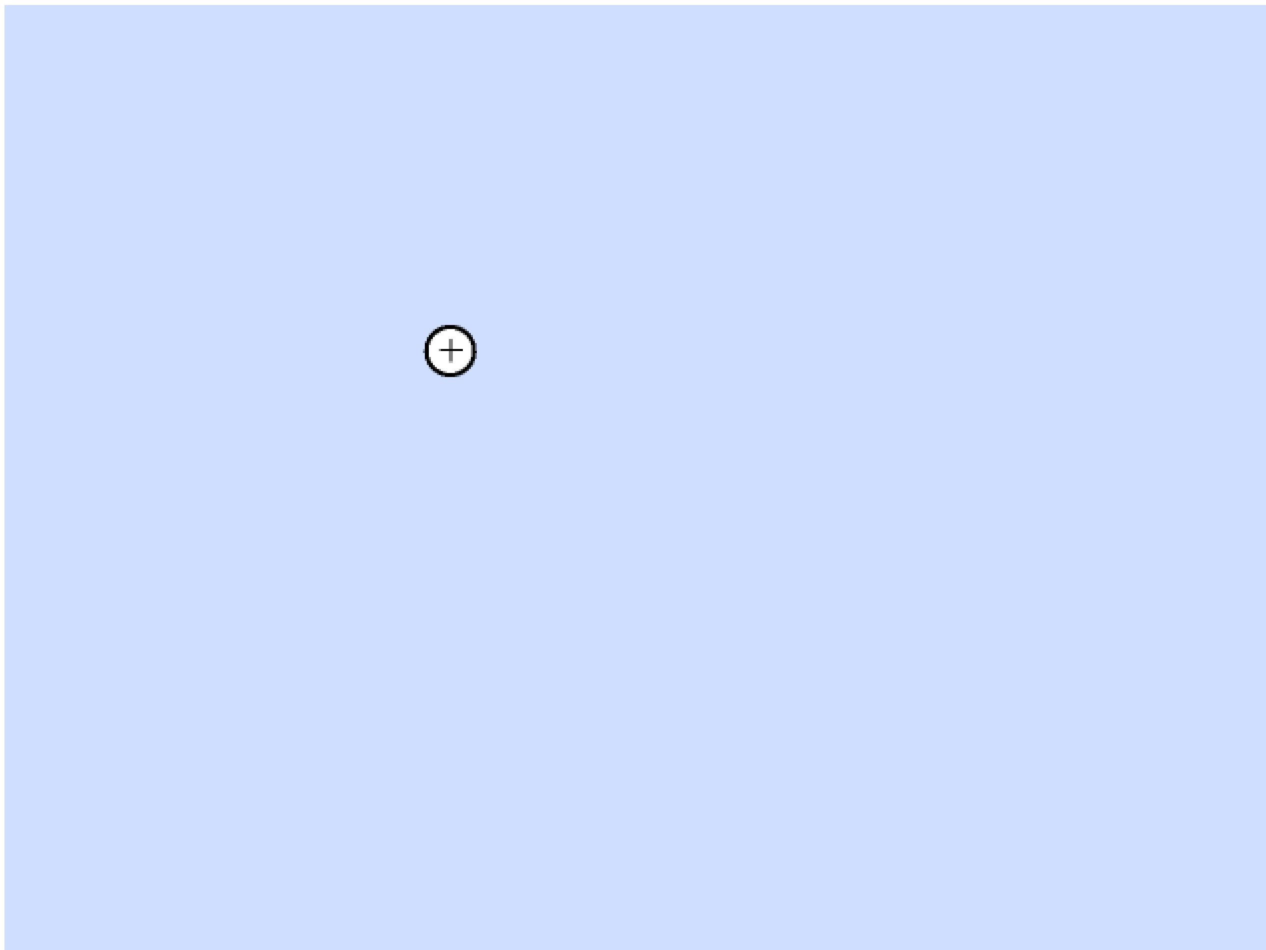
Genetic Programming

- 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.*

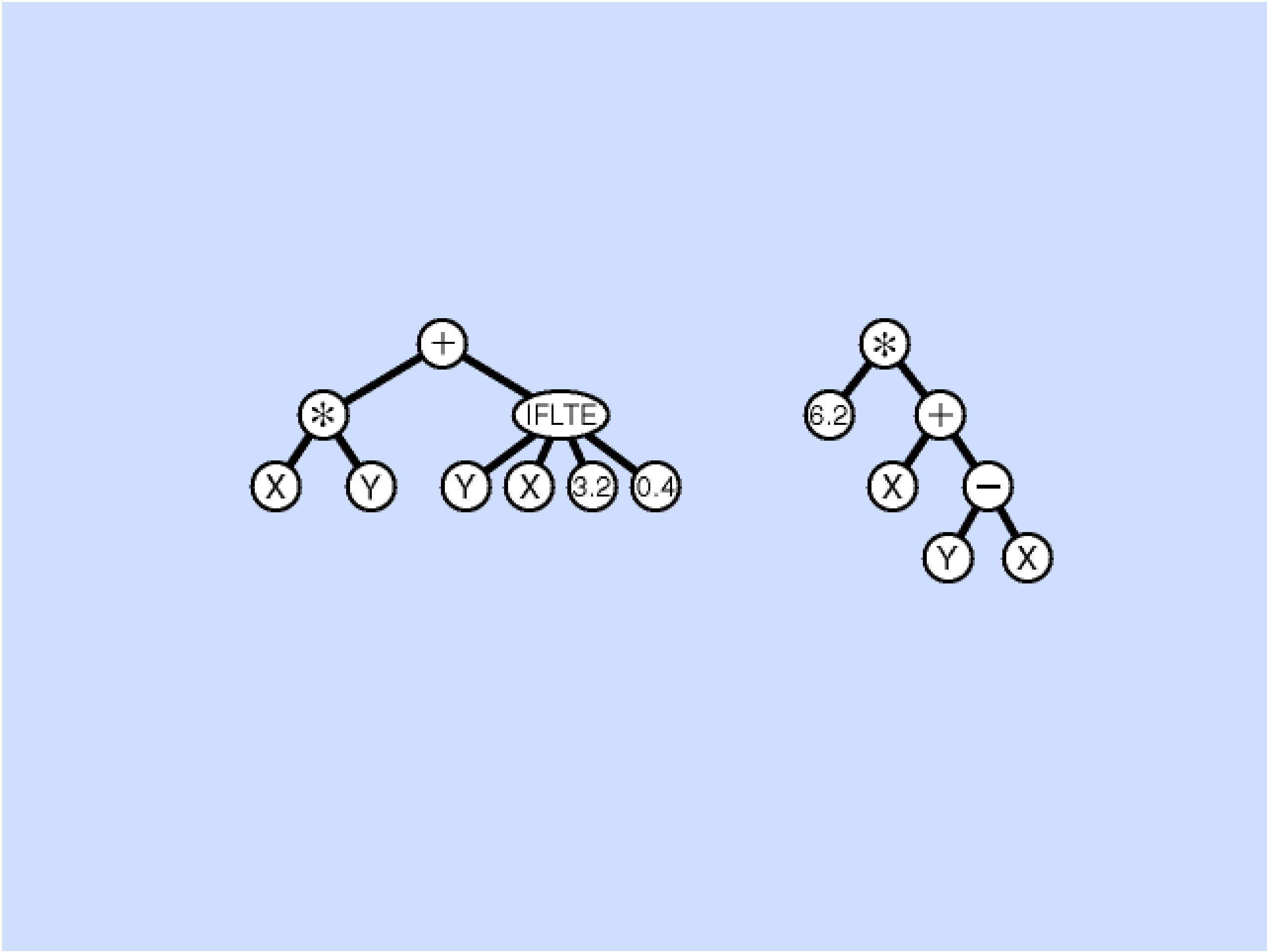
- applies Genetic Algorithms to program trees
  - Mutation and Cross-over adapated to tree structures
  - special operations like
    - inventing/deleting a subroutine
    - deleting/adding an argument,
    - etc.
- Several successful applications
  - 36 cases where it achieve performance competitive to humans
    http://www.genetic-programming.com/humancompetitive.html
- More information at http://www.genetic-programming.org/

Random Initialization of Population



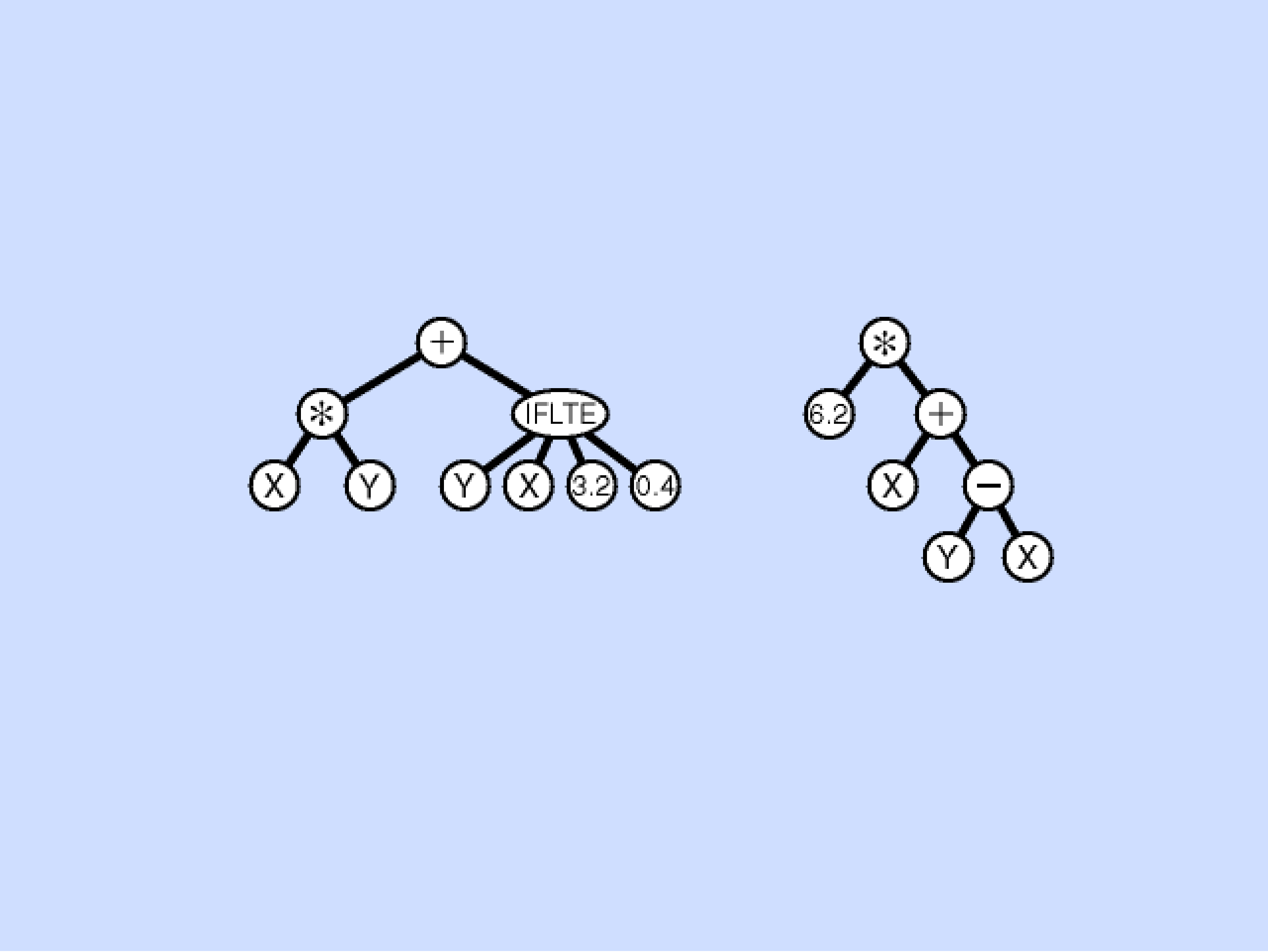
Animated Image taken from http://www.genetic-programming.com/gpanimatedtutorial.html

Mutation



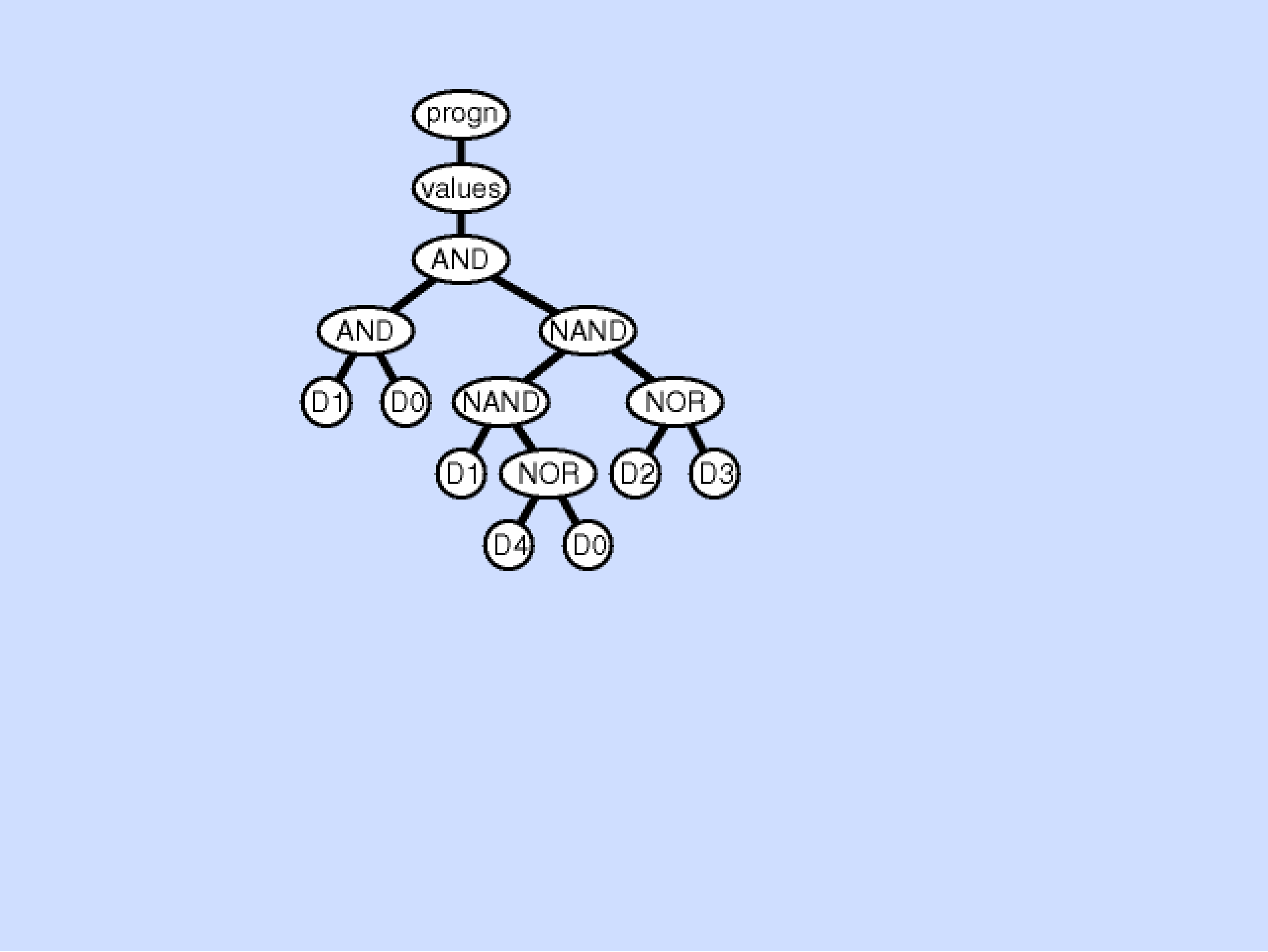
Animated Image taken from http://www.genetic-programming.com/gpanimatedtutorial.html

Cross-Over



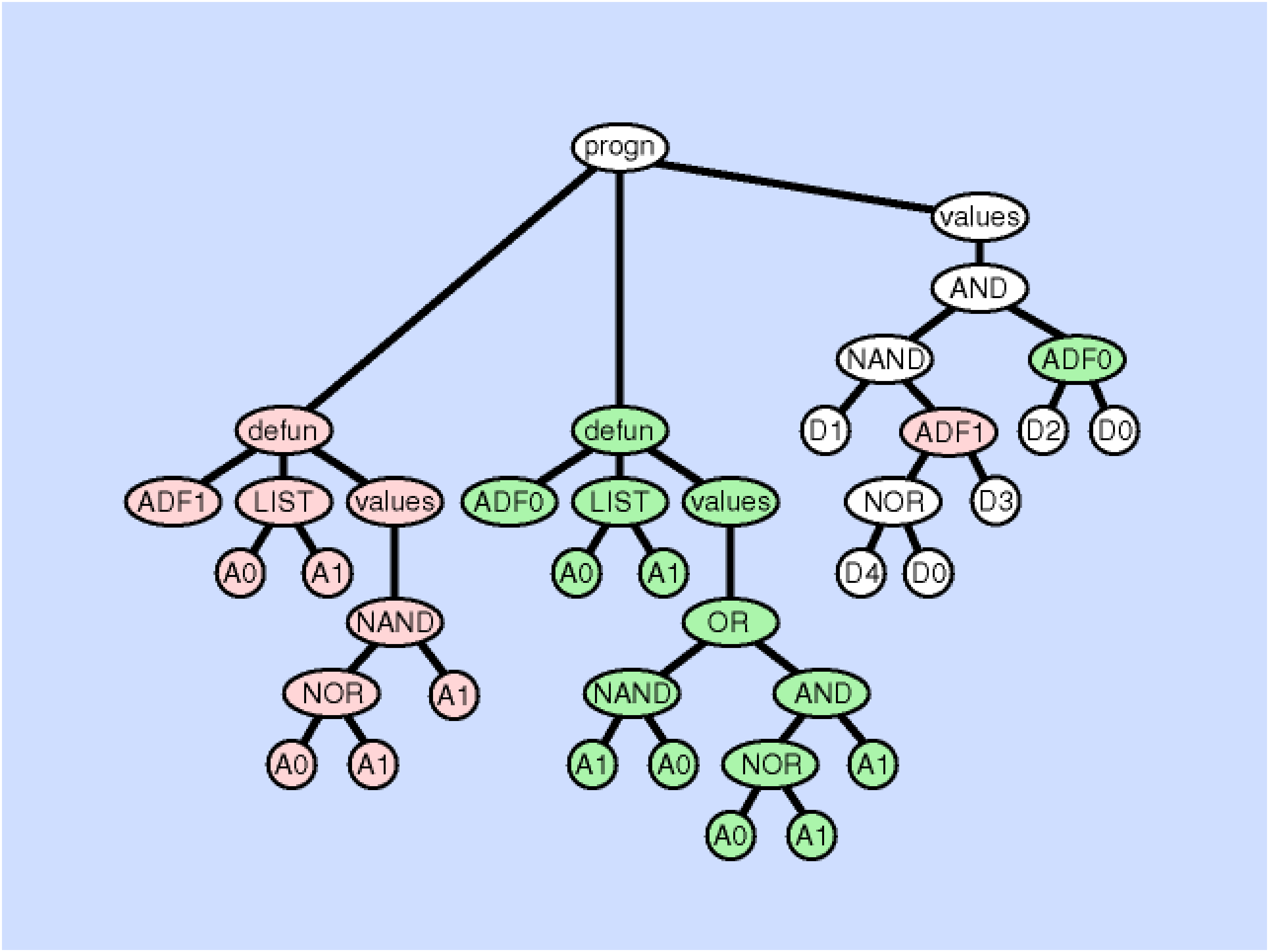
Animated Image taken from http://www.genetic-programming.com/gpanimatedtutorial.html

Create a Subroutine



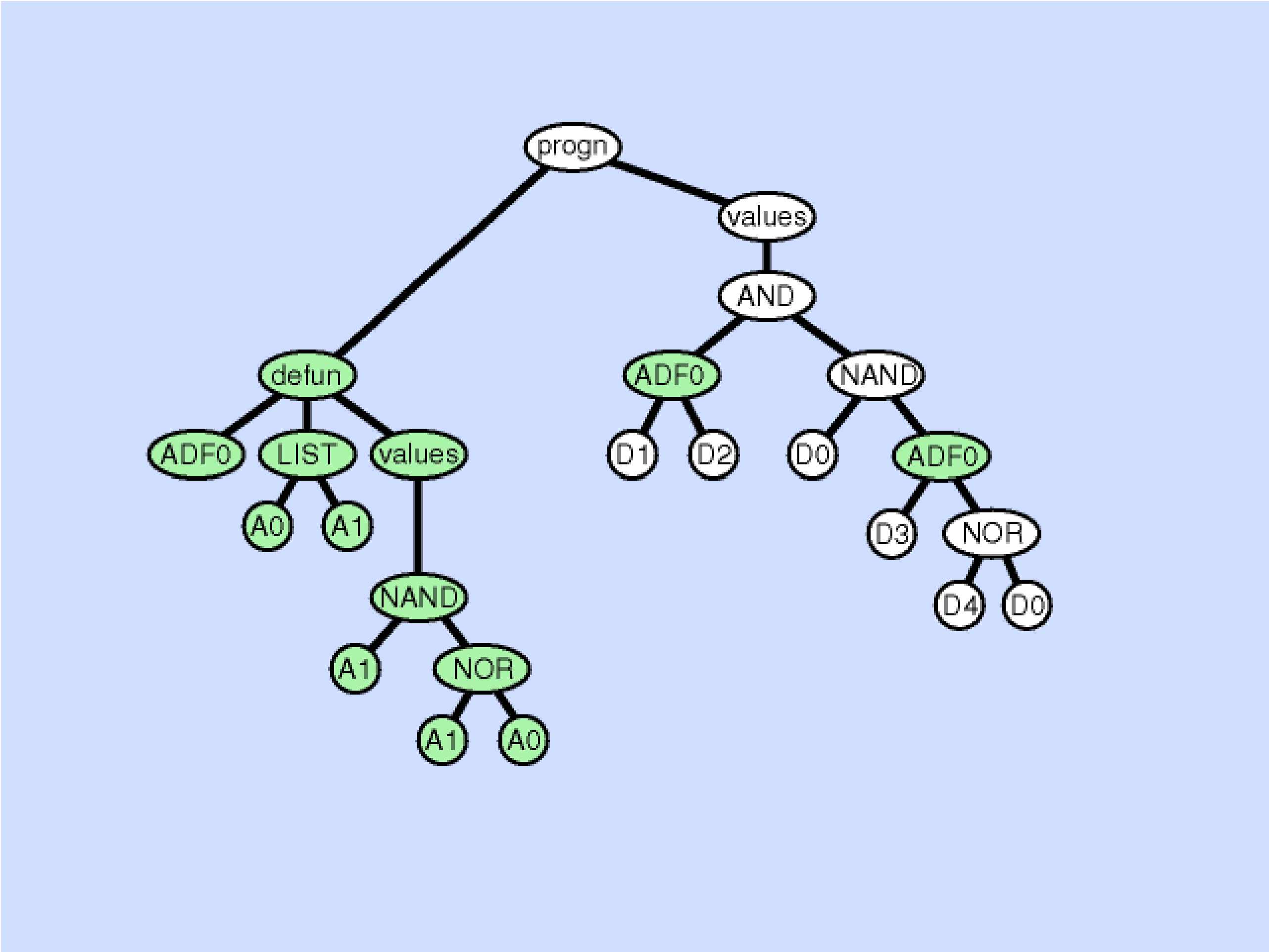
Animated Image taken from http://www.genetic-programming.com/gpanimatedtutorial.html

Delete a Subroutine



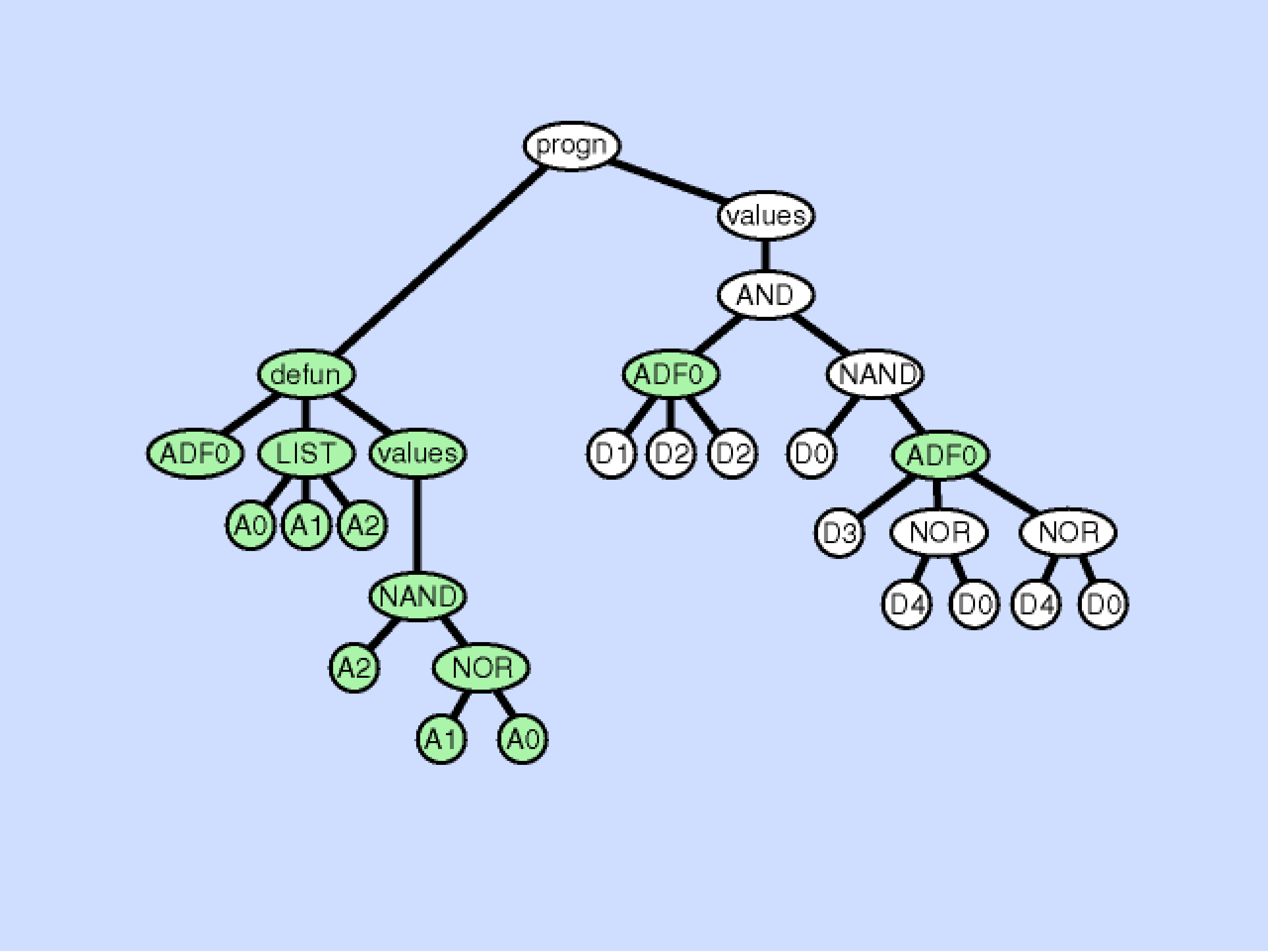
Animated Image taken from http://www.genetic-programming.com/gpanimatedtutorial.html

Duplicate an Argument



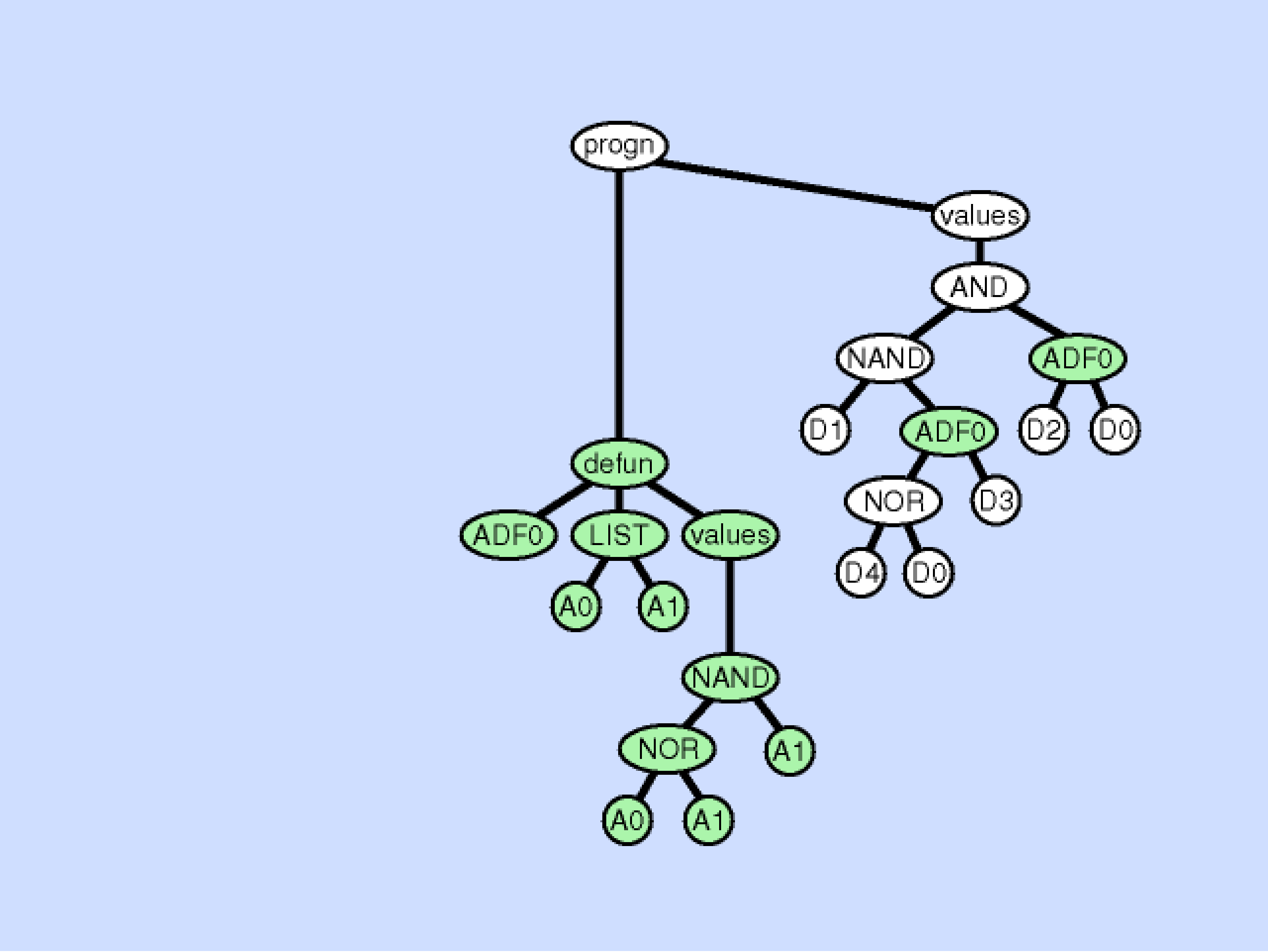
Animated Image taken from http://www.genetic-programming.com/gpanimatedtutorial.html

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

Local Search in Continuous Spaces

In many real-world problems the state space is continuous

- Discretize the state space
  - e.g., assume only n different positions of a steering wheel or a gas pedal
- Gradient Descent (Ascent)
  - hill-climbining using the gradient of the objective function f
  - f needs to be differentiable
- Empirical Gradient
  - empircally evaluate the response of f to small state changes
  - same as hill-climbing in a discretized space