Local Search and Genetic Algorithms

Dr. Demetrios Glinos
University of Central Florida

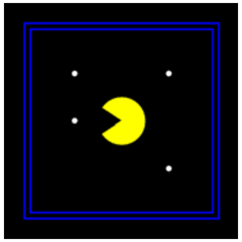
CAP4630 - Artificial Intelligence

Today

- Local Search
- Hill Climbing
- Simulated Annealing
- Beam Search
- Genetic Algorithms

Classic Search Problems Revisited

- Recall how we first defined a search problem
  - state space
  - successor function
  - start state
  - goal test
  - solution is a *sequence of actions (the plan)* that tranforms the start state into a goal state



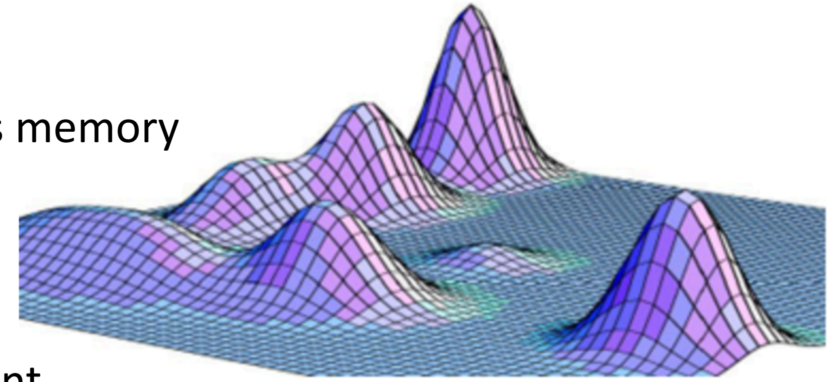
- We could use tree search algorithms to find optimal solutions
  - e.g., BFS, UCS
- We could also use heuristic algorithms
  - e.g., Greedy, A*

Optimization Problems

- Consider
  - state space too large to exhaustively enumerate
  - no particular start state
  - we don't know if we are in a goal state
  - path to a goal state doesn't matter, as long as we get there
  - optimal solution desirable, but not required
- This is the domain of many optimization problems
  - e.g., manufacturing process optimization, TSP
- Local search methods can often find reasonable solutions

Local Search

- Recall: Tree search keeps still-viable unexplored alternatives on a fringe
  - ensures completeness, goal is to find any goal state
- Local search:
  - An optimization technique for seaching a space that cannot be exhaustively enumerated
  - Goal is to find the global maximum
  - Basic idea: Improve a single option until you can't make it better
- Since there is no fringe:
  - Generally, much faster and uses less memory
  - BUT, incomplete and suboptimal



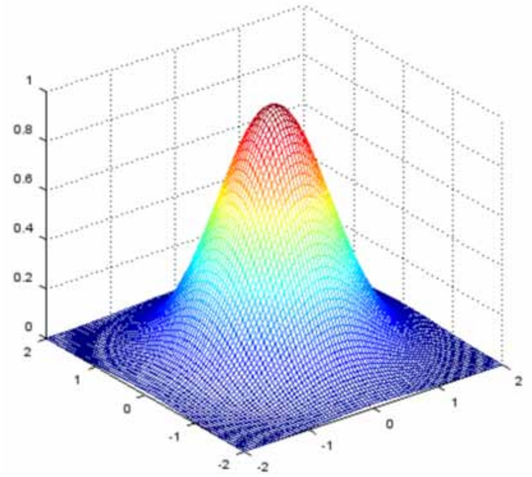
- Requirement: an evaluation function
  - to determine what is an improvement

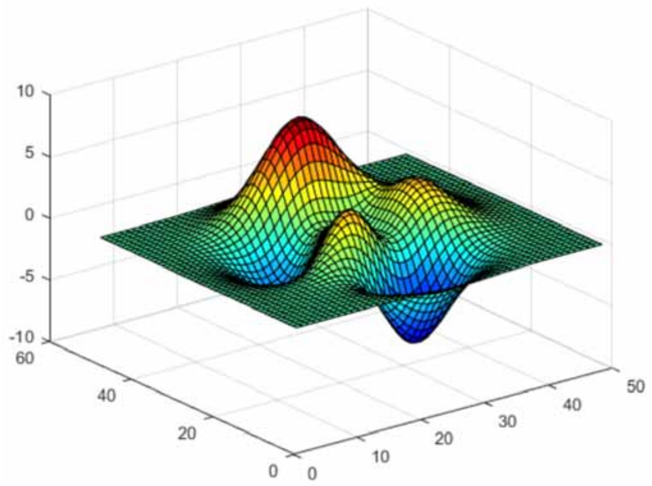
Today

- Local Search
- Hill Climbing
- Simulated Annealing
- Beam Search
- Genetic Algorithms

Hill Climbing

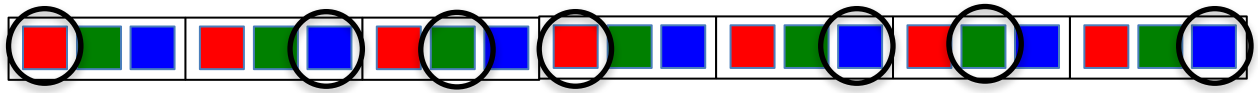
- **Hill climbing (aka greedy local search)**
- Basic idea:
  - Use an evaluation (heuristic) function for the states
  - Start anywhere
  - Move in direction of steepest gradient
  - If no neighbor better than current, then stop
- Can get stuck at local maximum
- Completeness and optimality not assured
- Nevertheless, can be useful in practice
  - when a "good" solution is good enough
  - when the search space is relatively well behaved





Example: CSP Min Conflicts

- Map-coloring CSP:

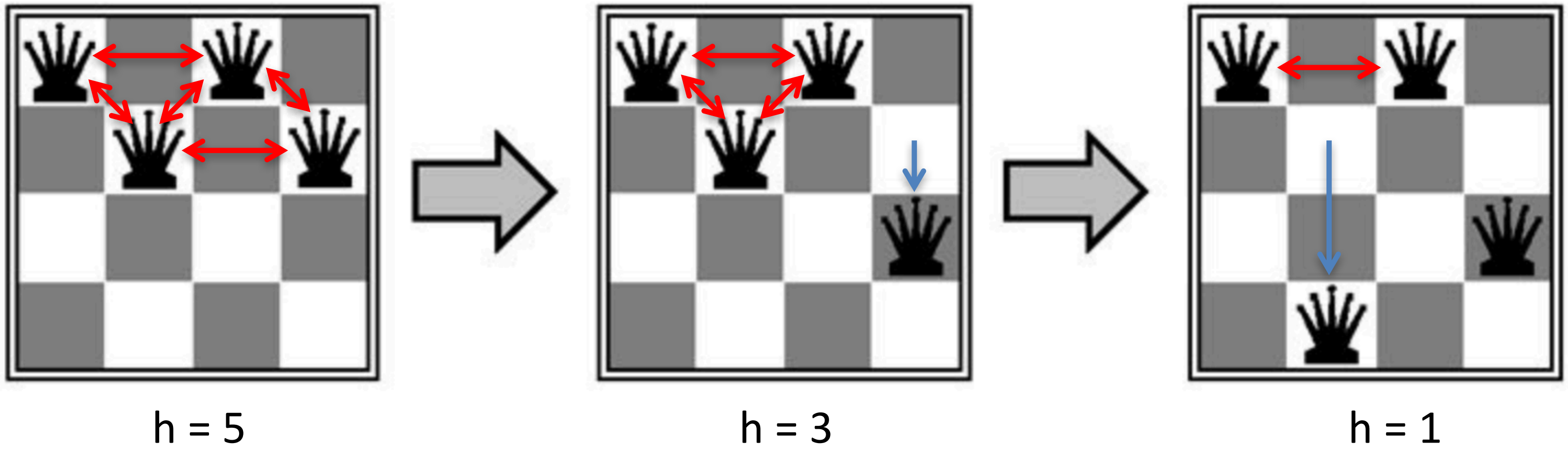


- Basic idea
  - Start with a complete assignment with unsatisfied constraints
  - Use operators to reassign variable values
  - Iterate until a solution found or exhaust all possibilities
  - Note: No fringe – work with just one assignment !

- Iterative Min Conflicts algorithm
  - Variable selection: random choice from among conflicting variables
  - “Min conflicts” value selection: choose value that results in fewest constraint violations

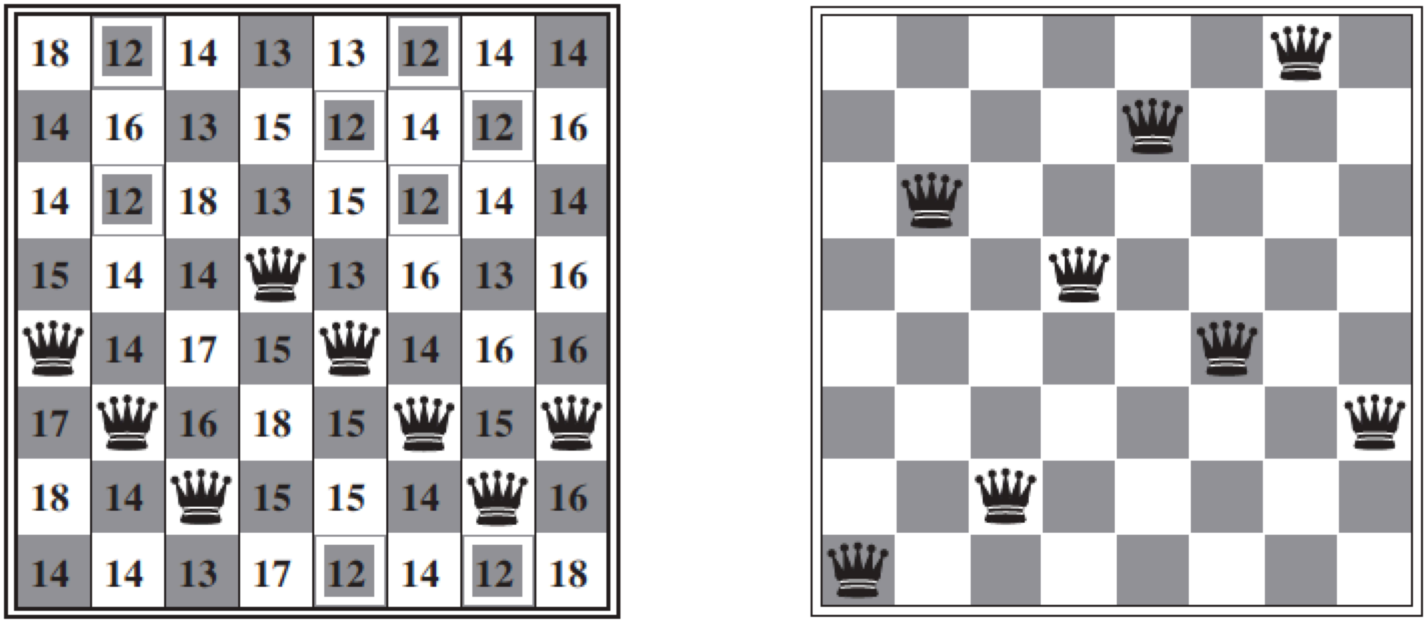
→ This is hill climbing with the function: heuristic h(n) = total number of constraints violated

Hill-Climbing Example: 4-Queens



- States:
  - 4 queens, 1 in each column (4^4 = 256 total states)
- Operator:
  - Move a queen vertically in its column
- Goal test:
  - No queen threatens another
- Heuristic:
  - h(n) = number of bidirectional attacks

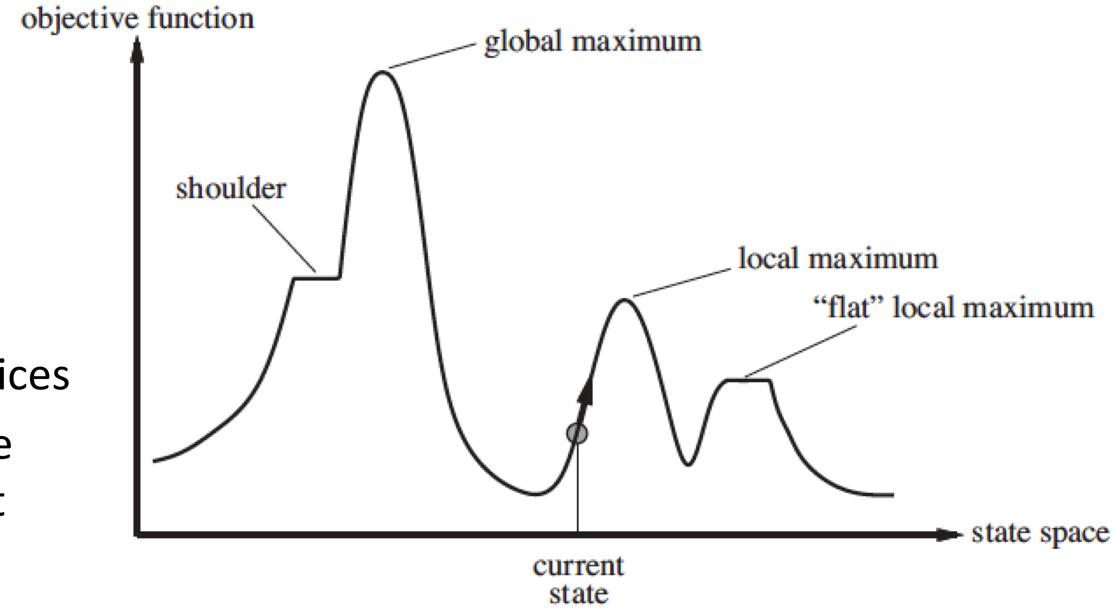
Example: Local Maximum



(a) (b) *source: Fig. 4.3*

- Consider: 8-queens problem, where each queen can move only within its column
- case (a): h = 17; best moves for each queen are marked
- case (b): a local maximum with h = 1 (not a goal state); every possible move has a higher cost

Improving Hill Climbing



- Basic hill climbing is steepest ascent
- Stochastic hill climbing
  - choose randomly from among several uphill choices
  - can weight choices by the steepness of the gradient
- First choice hill climbing
  - generate successors until find one that improves cost, and then take it
  - suitable where a state can have many (e.g., thousands) of successors
- Random restart hill climbing
  - perform as many climbs as needed until find an acceptable solution
  - each climb starts at a different random state
  - number of trials needed = 1/p, where p = prob of success for 1 trial
    - e.g., for 8-queens (column moves only), p ≈ 0.14, so need roughly 7 trials

Today

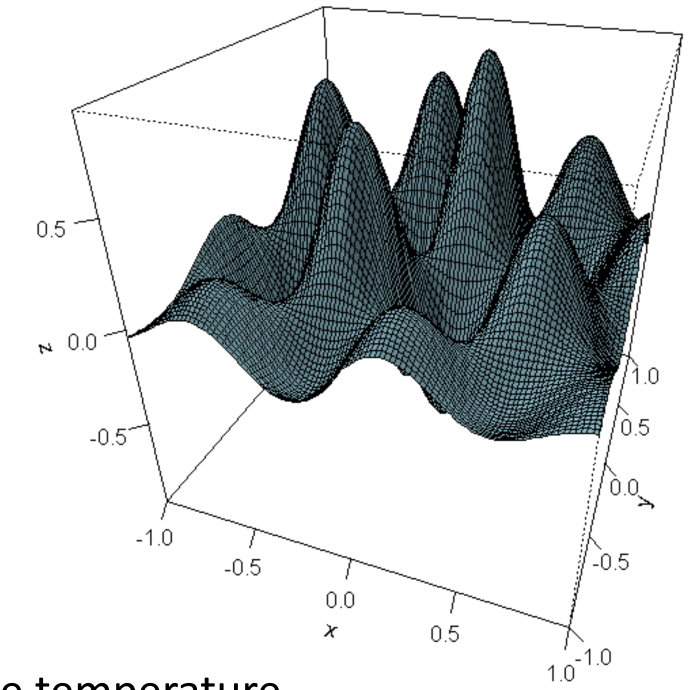
- Local Search
- Hill Climbing
- Simulated Annealing
- Beam Search
- Genetic Algorithms

Simulated Annealing

Annealing , in metallurgy, involves heating and slowly cooling a metal, allowing it to form a more regular crystalline structure, which allows the metal to be deformed and to be shaped more easily

Simulated Annealing

- Purpose: to escape local maxima
- How: incorporate a temperature parameter into hill climbing
  - at high temperatures: more exploration
  - at low temperatures: more exploitation
  - use a cooling schedule to slowly lower the temperature



Basic Algorithm for Simulated Annealing

Goal: find the global *minimum*

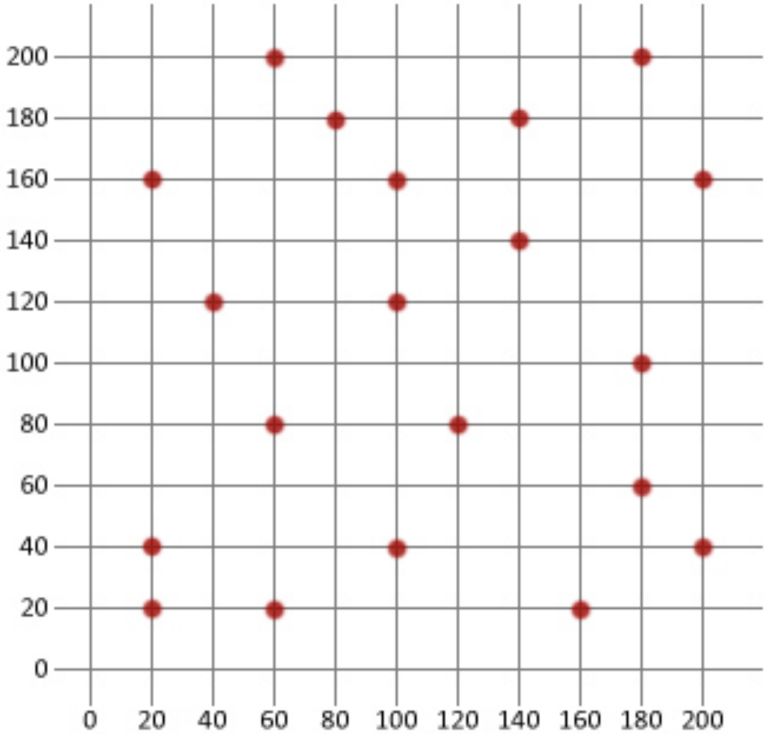
1. Start with a candidate solution
2. Calculate its value using some cost function
3. Generate a neighboring solution
4. Calculate the neighboring solution's cost
5. Compare them:
    a. If $cost_{new} < cost_{old}$, accept the new solution
    b. If $cost_{new} > cost_{old}$, maybe accept the new solution, depending on the temperature
6. Repeat steps 3 to 5 above, slowly reducing the temperature, until convergence or reach a max limit on iterations

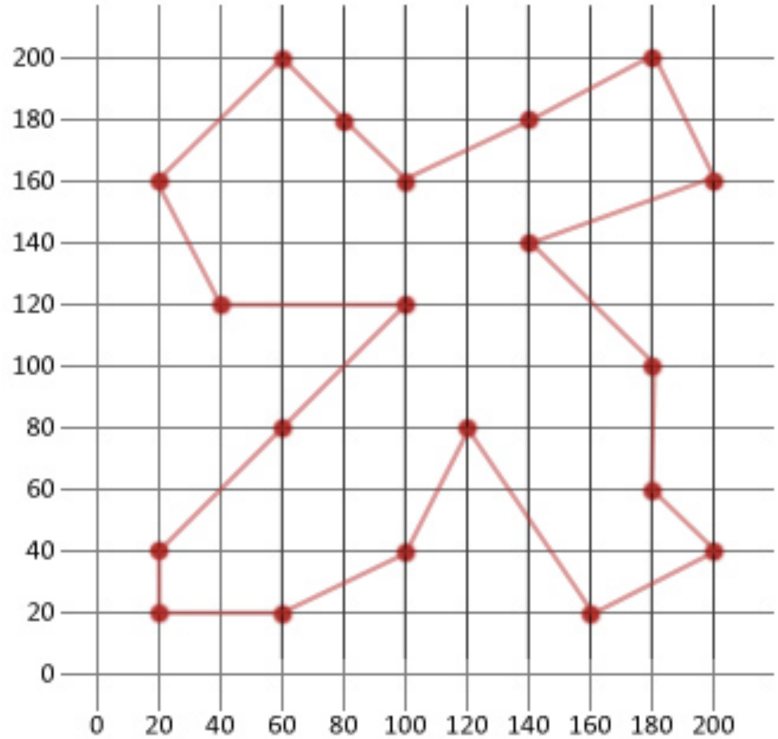
Simulated Annealing Details

- Initial candidate:
  - can be truly random
- Cost function:
  - can be anything appropriate (e.g., total distance traveled for TSP)
- Neighboring soluton:
  - must differ from current only slightly (must be a neighbor in the search space)
  - choose randomly among the neighbors
- Always accept a lower cost candidate: $\Delta Cost = cost_{new} - cost_{old} < 0$
- Acceptance function for higher-cost candidate:
  - Metropolis-Hastings algorithm: accept if $e^{-\Delta Cost/Temp} > Random(0, 1)$
  - threshold acceptance algorithm: accept if $\Delta Cost < threshold$
- Temperature schedule:
  - Start temp at value 1.0 and reduce by α every n iterations
  - Typically, $0.8 < \alpha < 0.99$
  - $Temp_i = \alpha^i$

Example: Simulated Annealing

Sweet spot: Combinatorial optimization, such as Traveling Salesperson Problem (TSP)

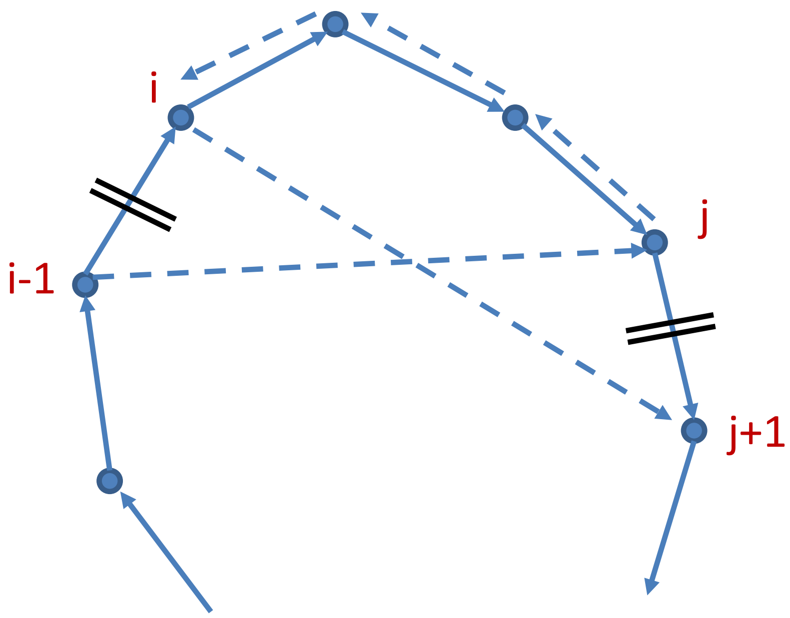




Search space for 20-city TSP is 20! = 1.55×10^{25}

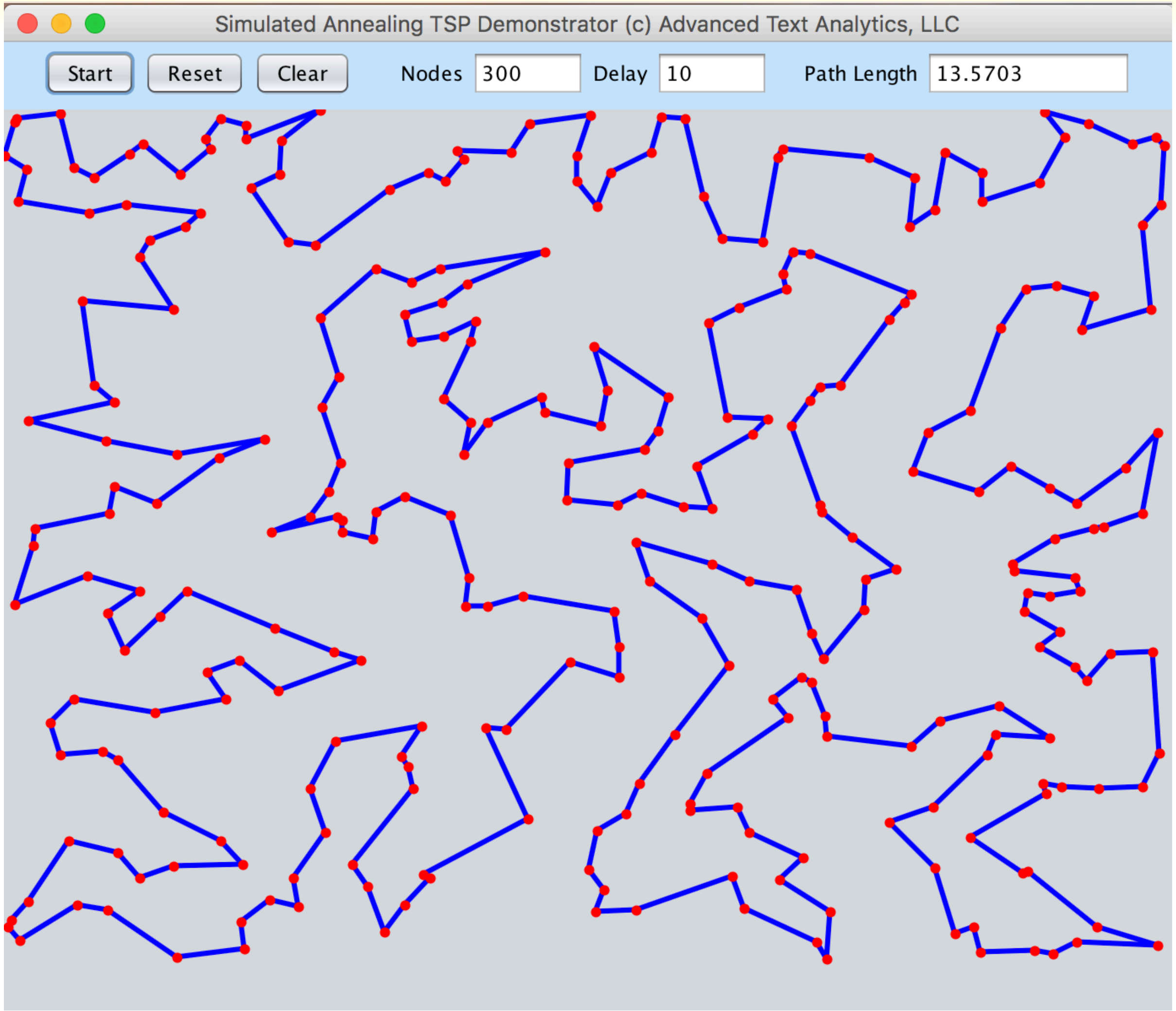
Demo: Simulated Annealing TSP

- Random locations for nodes
- Random initial sequence
  - complete Hamiltonian circuit
- Neighboring solution selection:
  - choose 2 random nodes
  - reverse the path between them and tie into full path
  - compare total cost for revised sequence to current cost
- Cost function: total Euclidean path length
- Cooling rate: α = 0.9999



before: solid arrows
after : dashed arrows
cuts as shown

Demo: Simulated Annealing TSP



demo: SimulatedAnnealing.jar

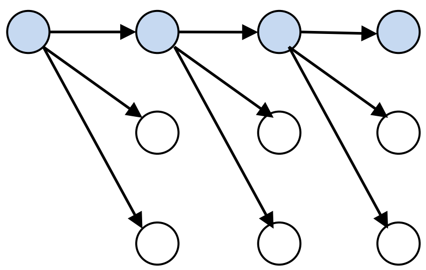
Today

- Local Search
- Hill Climbing
- Simulated Annealing
- Beam Search
- Genetic Algorithms

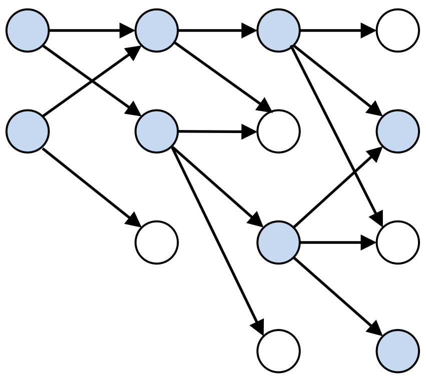
Beam Search

- Like local greedy search, except
  - start with K random initial states
  - at each step, generate all successors of all K states
  - but keep only the K "best" states for next iteration

Local greedy Search



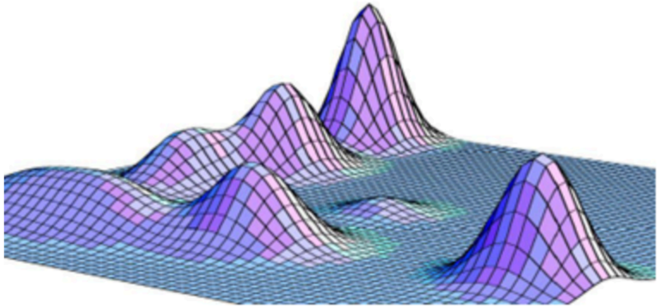
Beam Search



Q: How is this different from having K random restarts?

Improving Beam Search

- Basic beam search
  - choose the "best" k successors from the pool of candidate successors
  - can suffer from the lack of diversity among the k states
  - population can be concentrated in a local region of the state space



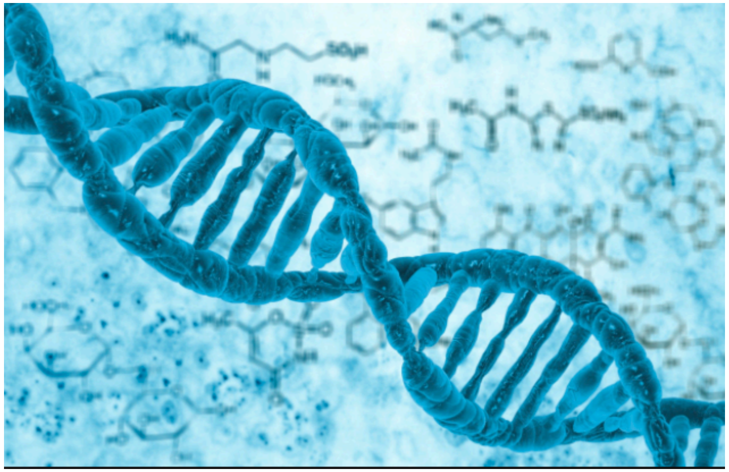
- Stochastic beam search
  - choose a random set of k successors
  - weight each choice according to its value
  - analogy to natural selection
    - population at next step consists of some subset of the successors (offspring) of the current state (population organism)

Today

- Local Search
- Hill Climbing
- Simulated Annealing
- Beam Search
- Genetic Algorithms

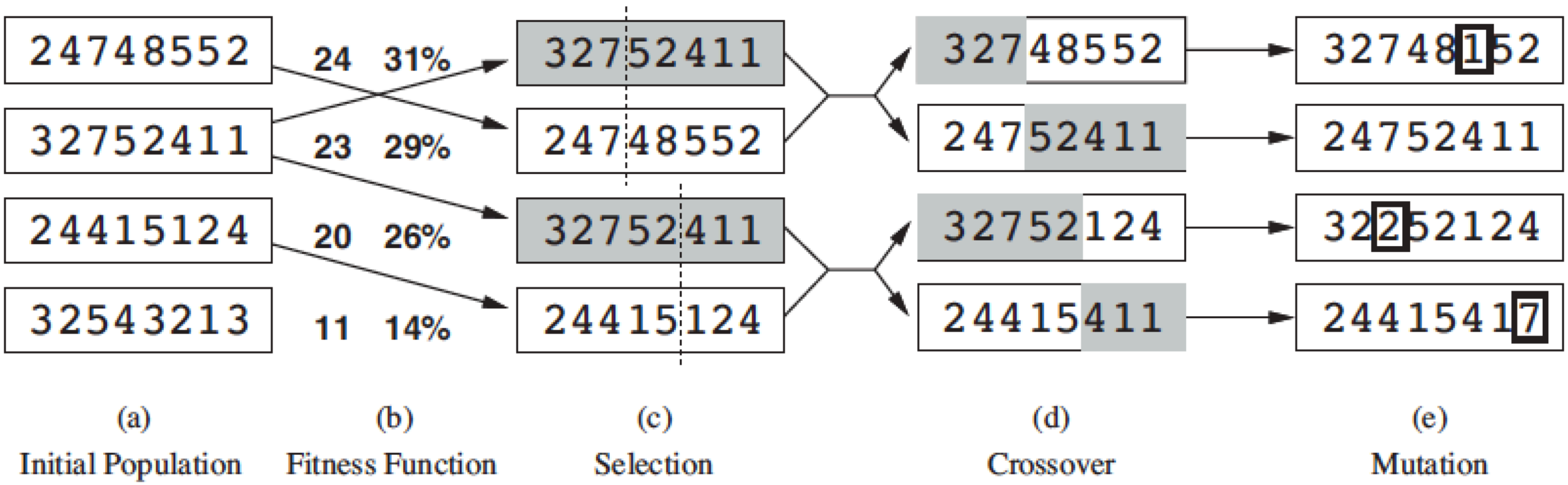
Genetic Algorithms

- Based on genetics in biology
  - Chromosomes encode traits
  - Offspring inherit "chromosomes" from both parents
  - Mutations can occur
  - Some mutations more successful than others



- **Genetic Algorithms**
  - encode candidate solutions as a population of ***chromosomes***
  - use genetics-inspired operators to generate offspring
    - selection – of parents
    - crossover – inheriting from both parents
    - mutation – random changes in components of candidate solutions
  - evaluate offspring according to a fitness function
  - replace the population with the next generation

GA Details

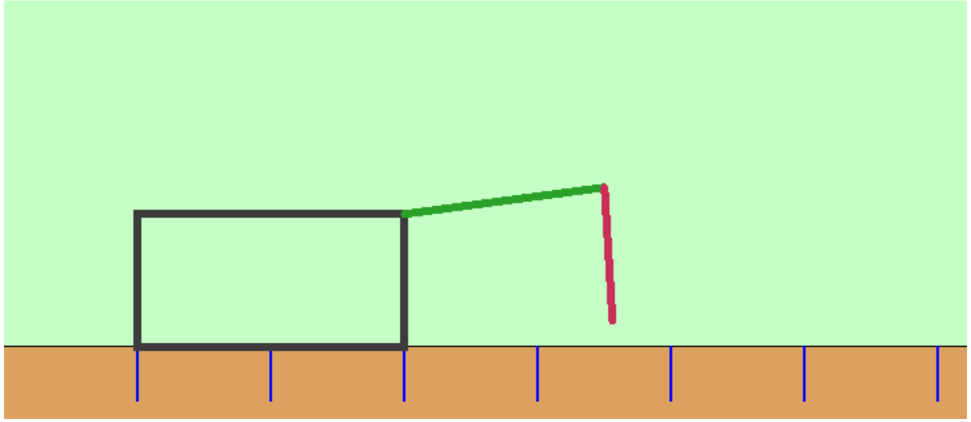


- Chromosomes are generally sequences of symbols, that can represent
  - solutions of complex functions
  - finding sequences of amino acids that will fold to a desired 3-D protein structure
  - sequences of motion for a robot crawler
  - whether or not all the same length depends on the problem

GA Details

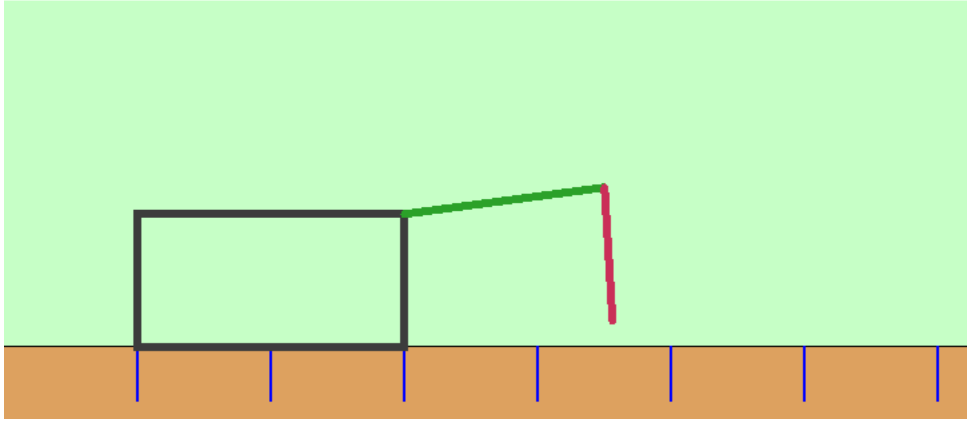
- Selection
  - fitness-proportionate - the more fit a chromosome, the more likely it will be selected as a parent
  - generally done with replacement (same chromosome can be parent to > 1)
- Crossover
  - usually at 1 randomly-chosen point, with uniform probability p_c
  - point can be different on each chromosome
  - multi-point crossover also possible
- Mutation
  - random change to a gene in each chromosome, with probability p_m
  - can restrict to 1 or >1 mutation per chromosome
- Fitness function
  - custom for each problem
- Next Generation
  - can select top n most fit chromosomes
  - can propagate elites (the top few always make it to the next generation)

GA Demo: Robot Crawler



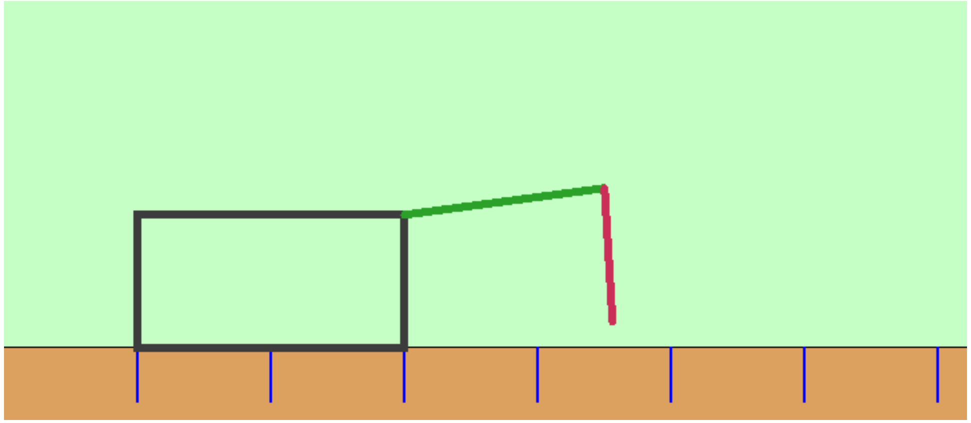
- State space
  - discretized
  - green upper arm: +/- 45 degrees from horizontal
  - red lower arm: +/- 45 degrees from perpendicular to green arm

GA Demo: Robot Crawler



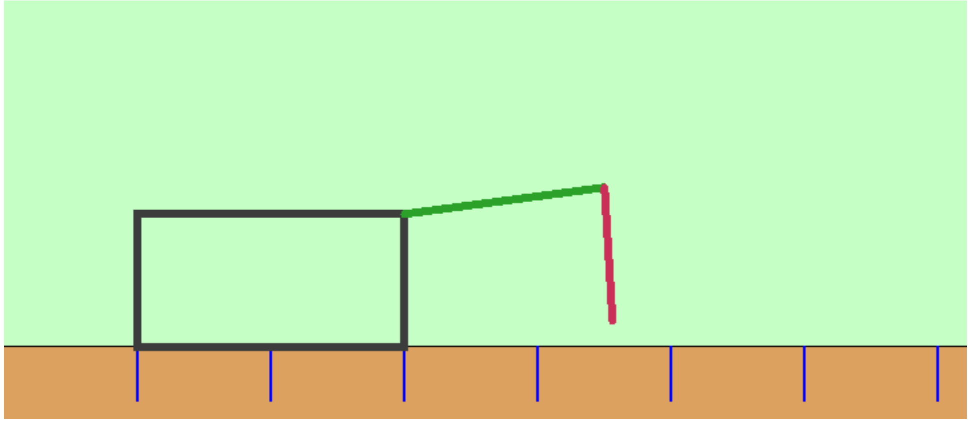
- Chromosome
  - start with green horizontal and red vertical, touching ground
  - random sequence of NSEW movements
  - do not need all to be same length
  - must validate for
    - movement out of range
    - loops, including stutter

GA Demo: Robot Crawler



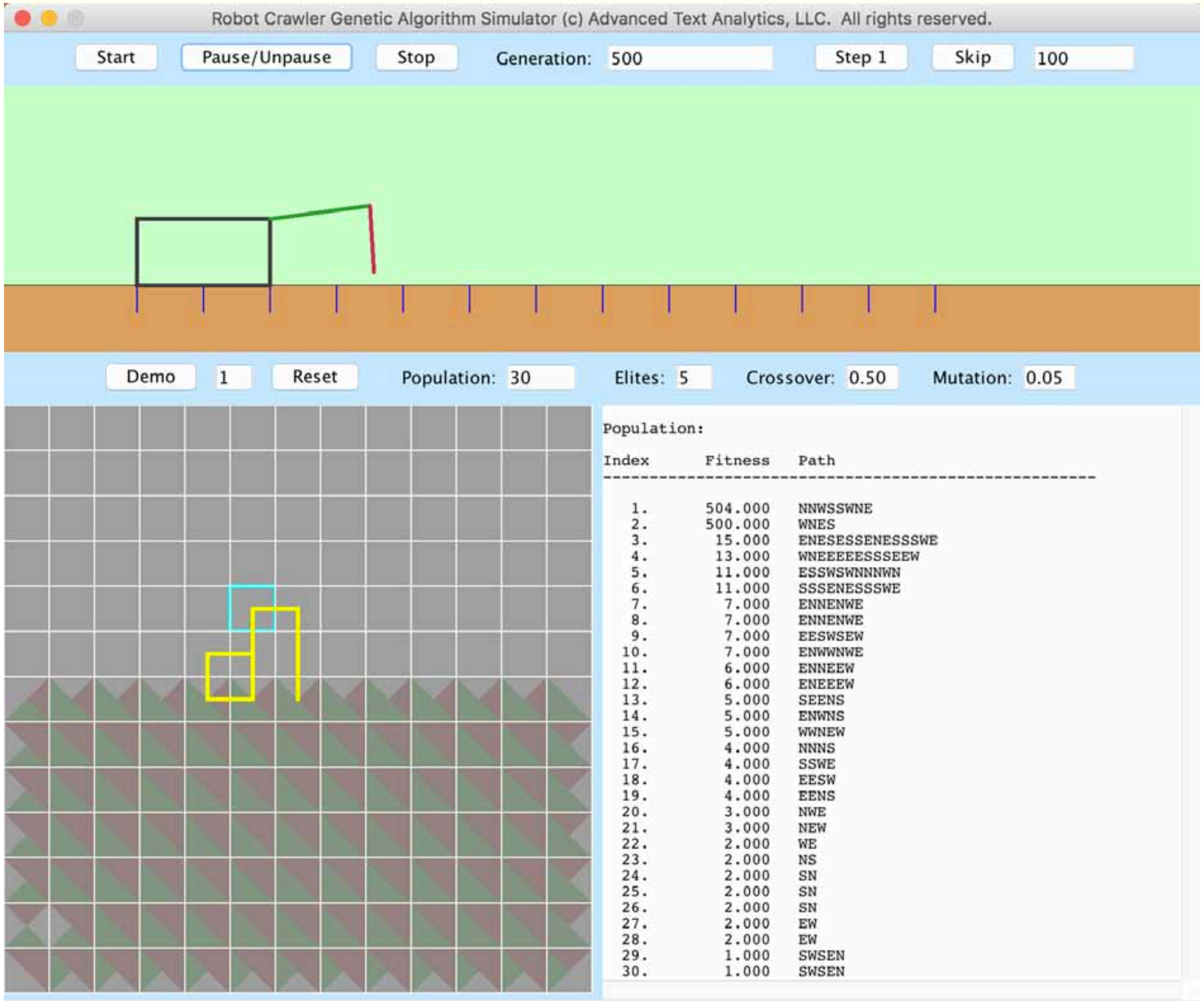
- Fitness
  - wish to reward forward motion +500, backward -500, within a loop
  - “forward motion” defined to be left corner to the right of previous spot with right corner on ground
  - also reward longer sequences, so +1 for every element of sequence outside of a loop

GA Demo: Robot Crawler



- Crossover
  - Single point, must validate
- Mutation
  - Single point, must validate
- Population
  - Start with 30
  - Use elites

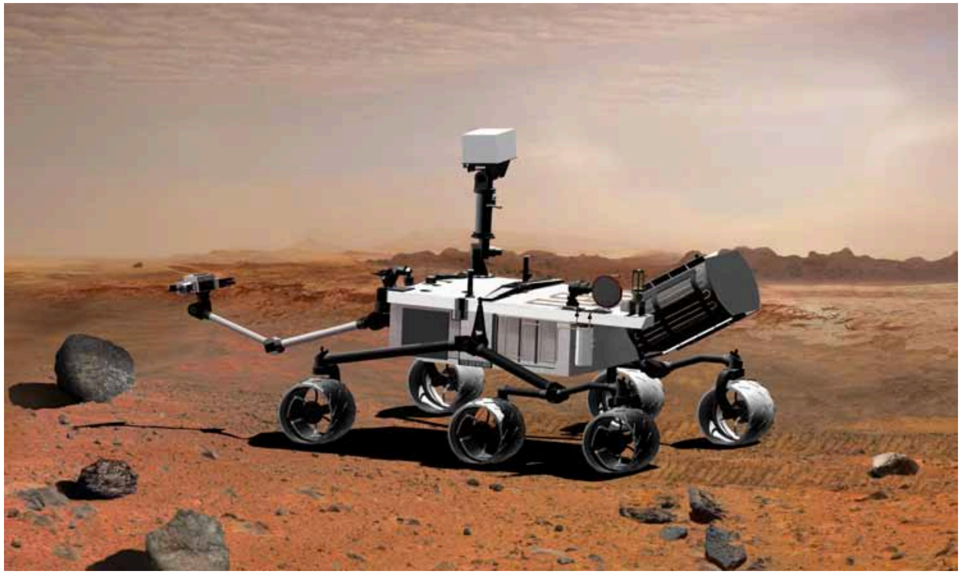
GA Demo: Robot Crawler



demo: CrawlGA

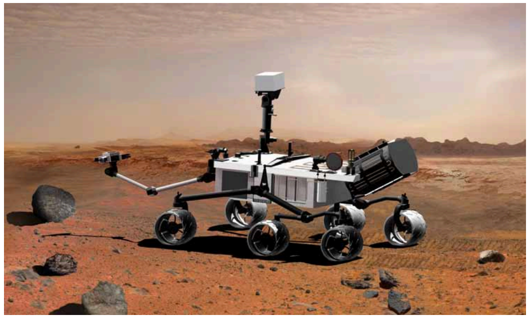
GA Problem: Rover Path Planner

- Consider a planetary rover
  - current position known
  - goal position determined externally
  - sensors on board
  - obstacles on path to goal position



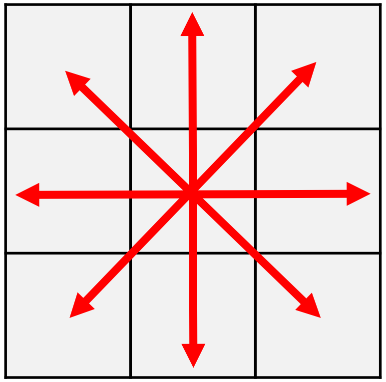
- Goal
  - find a safe path to goal position

Path Planner Design



- State space
  - discretized
  - use Cartesian grid
  - could also use hexagonal grid, etc.

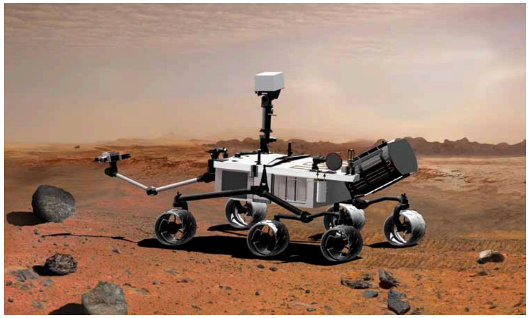
- Chromosome
  - sequence of moves
  - must avoid obstacles



- Move
  - vector representation: direction + length
  - 8 directions
  - endpoint must "snap" to center of a grid cell

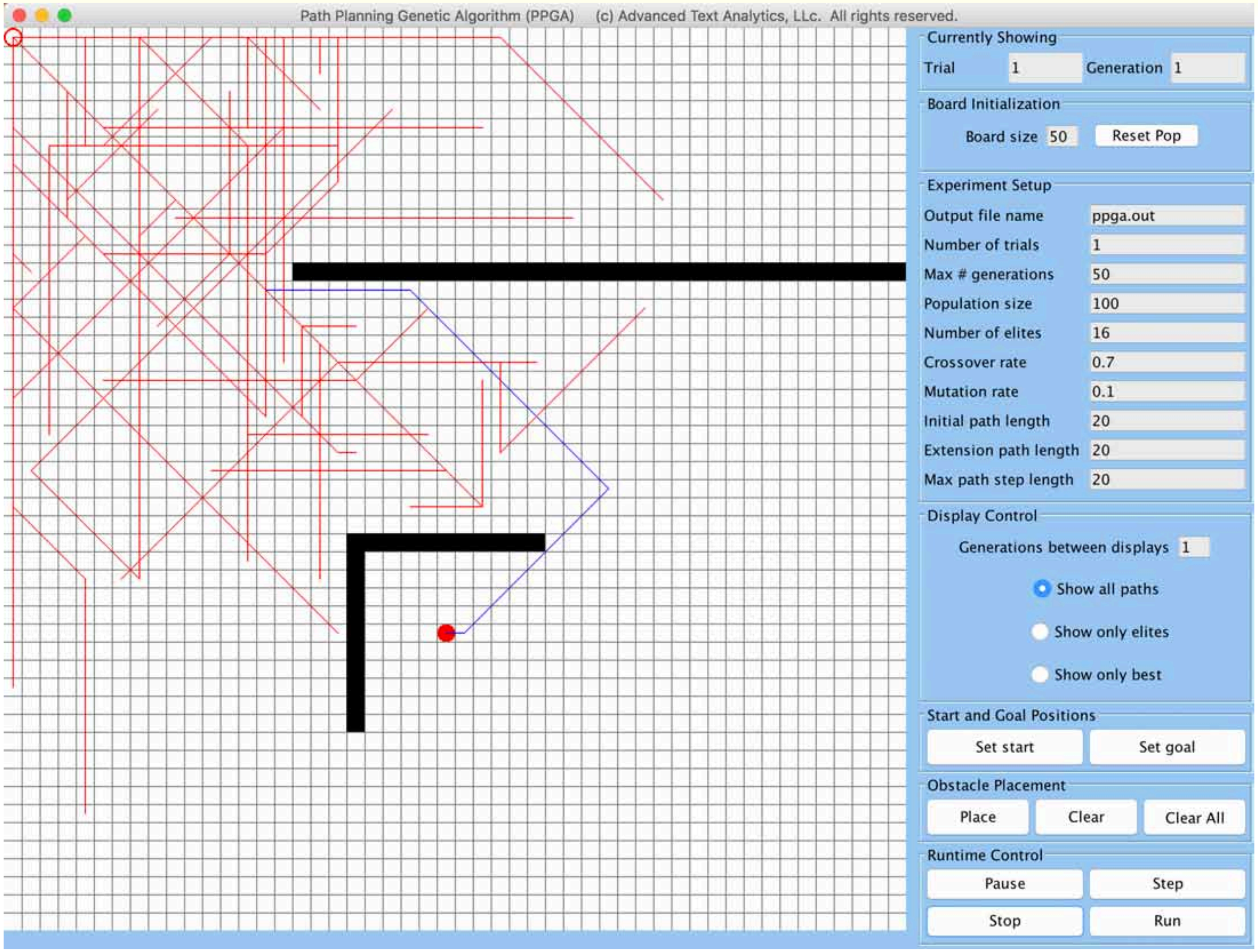
Path Planner Operators

- Mutation
  - add random number (bounded) of "legs" to current path
  - each leg can be in 1 of 8 possible directions
  - each leg can be a random length (bounded)
  - prune path if bumps into wall or goes off the board
- Crossover
  - single point
  - prune both paths as appropriate



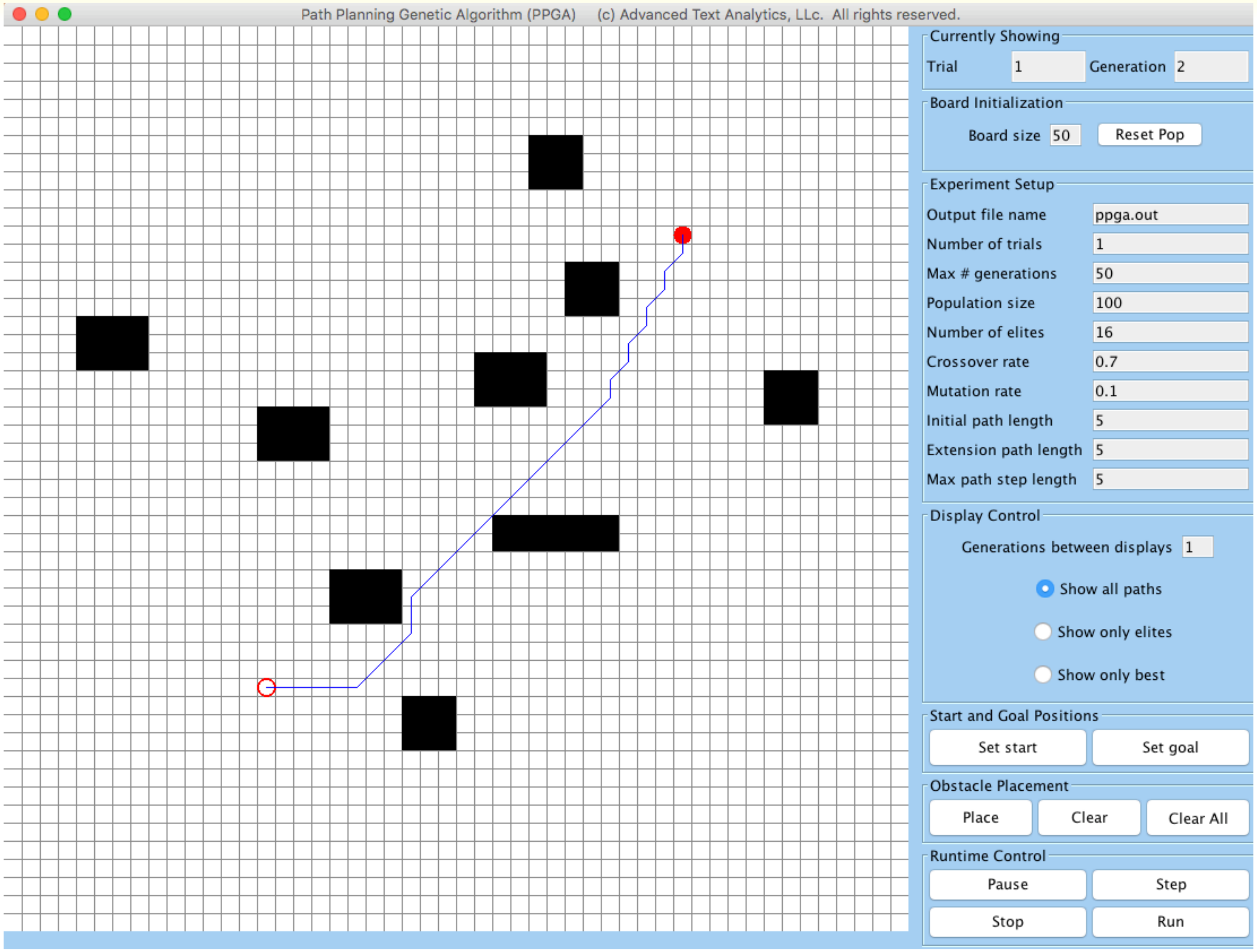
- Fitness function
  - use heuristic: Euclidean distance to goal
- Tuning parameters
  - crossover rate, random mutation and crossover parameters

GA Demo: Rover Path Planner



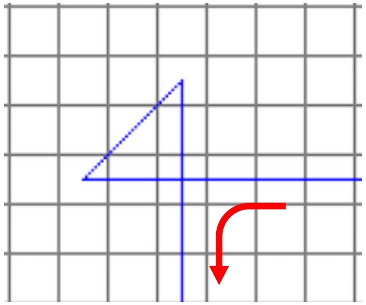
demo: PPGA.jar

GA Demo: Rover Path Planner

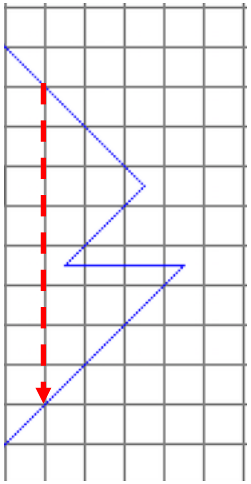


demo: PPGA.jar

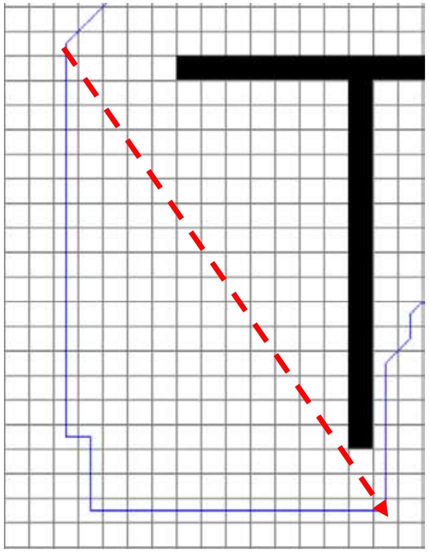
Improving the Path Found



Cut out loops



Straighten out bights



Cut corners

Note: Can also choose key points in the path and fit a smooth curve (e.g., spline)