AI Programming

CS662-2013S-08

Local Search / Genetic Algorithms

David Galles

Department of Computer Science
University of San Francisco

08-0: Overview

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

08-1: Local Search

- So far, stored the entire path from initial to goal state
- Path is essential – the path *is* the solution
  - Route finding
  - 8-puzzle
  - (to a lesser extent) adversarial search
- We know what the goal state is, but not how to reach it

08-2: Local Search

- For some problems, we don't care what the sequence of actions are – the final state is what we need
- Constraint Satisfaction Problems & Optimization Problems
  - Finding the optimial (or satisfactory) solution is what is important
  - 8-Queens, Map Coloring, Scheduling, VLSI layout, Cryptography
- The solution is an assignment of values to variables that maximizes some objective function
- We don't care *how* we get to the solution, we just need the values of the variables

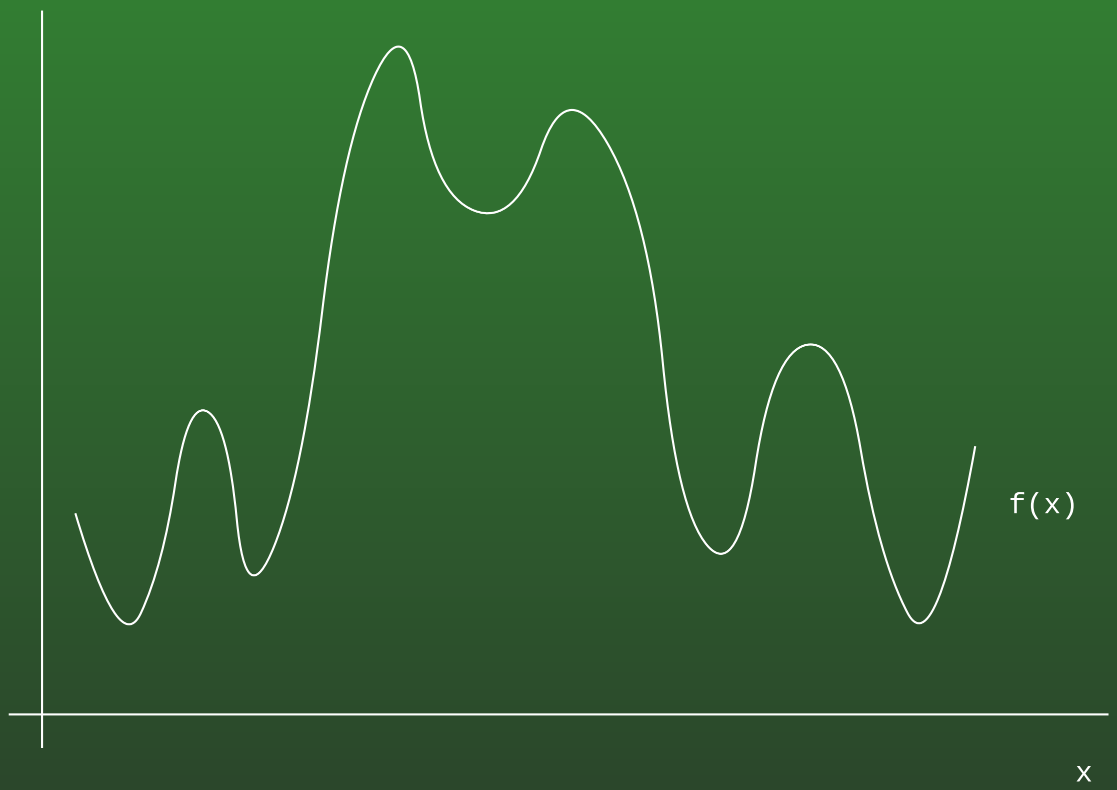
08-3: Local Search

- Search algorithm that only uses the current state (no path information) is a *local search* algorithm
- Advantages
  - Constant memory requirements
  - Can search huge problem spaces
- Disadvantages
  - Hard to guarantee optimality, might find only a local optimum
  - May revisit states or oscillate (no memory)

08-4: Search Landscape

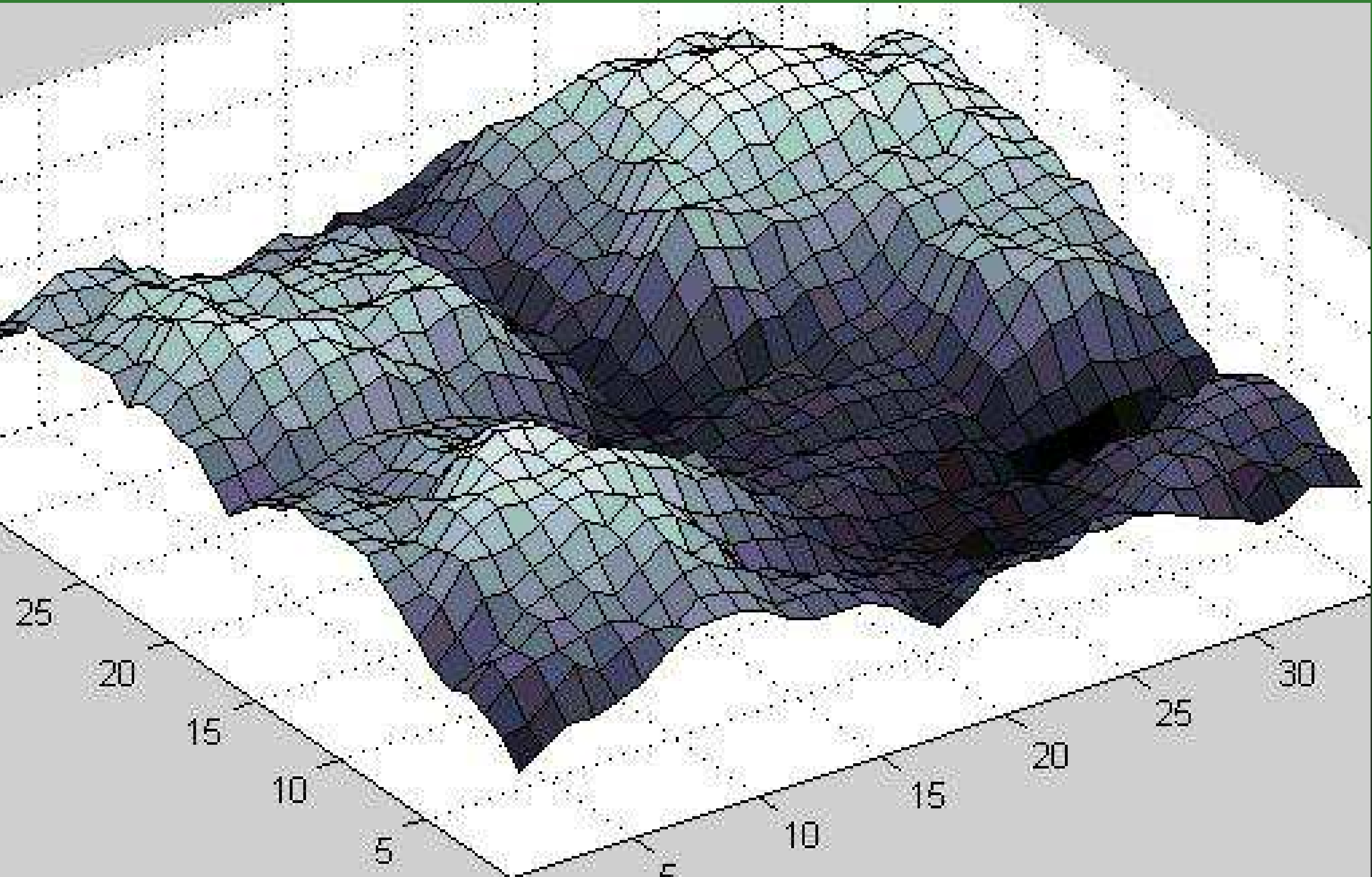
- Local search can be useful for optimization algoritms
- "Find parameters such that $o(x)$ is maximized/minimized"
- Search problem: state space is the combination of value assignments to parameters
- If there are n parameters, we can imagine an $n + 1$ dimensional space, where the first n dimensions are the parameters of the function, and the $n + 1$th dimension is the *objective function*
- *Search Landscape*
  - Optima are hills
  - Valleys are poor solutions

08-5: Search Landscape



- Maximize function $f(x)$

08-6: Search Landscape



08-7: Search Landscape

- Lanscapes are a useful metaphor for local search algorithms
- Visualize climbing a hill, or descending a valley
- Gives us a way of differentiating easy problems from hard problems
  - Easy: Few peaks, smooth surfaces, no ridges/plateaus
  - Hard: Many peaks, jagged or discontinuous surfaces. plateaus

08-8: Hill Climbing Search

- Simpliest local search: Hill Climbing
- At any point, look at all your successors (neighbors), move in the direction of greatest positive change
- Similar to Greedy Search
- Requires very little memory
- Stuck in local optimal
- Plateaus can cause aimless wandering

08-9: Hill Climbing Search

- Example: n-Queens
- Each position in the search space is defined by a n-unit vector
  - V[i] = column of row in position i
  - (examples on board)
- Function is the number of conflicts
- Trying to minimize function

Hill Climbing Search

- Find roots of an equation: $f(x) = 0$, f differentiable
- Guess and x_1, find $f(x_1)$, $f'(x_1)$
- Use tangent line to $f(x_1)$ (slope = $f'(x_1)$) to pick x_2
- Repeat: $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$
- Hill climbing search
- Works great on smooth functions

08-11: Hill Climbing Search

- Advantages to Hill Climbing
  - Simple to code
  - Requires little memory
  - May not need to do anything more complicated
- Making Hill Climbing better:
  - Stochastic hill-climbing – pick randomly from uphill moves
  - Weight probability by degree of slope

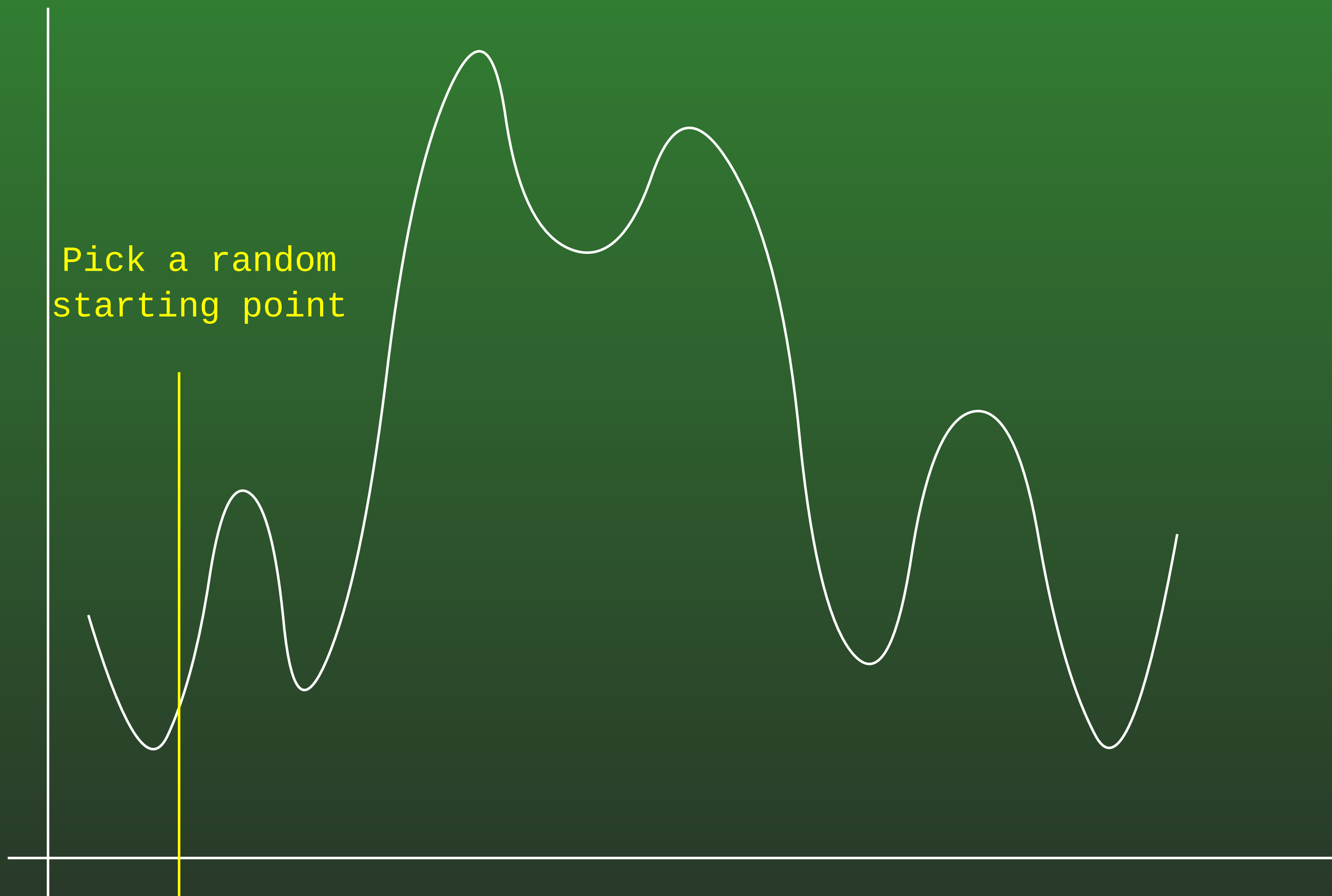
08-12: Improving Hill Climbing

- Random-Restart Hill-Climbing
- Run Hill Climbing until an optimum is reached
- Randomly choose a new initial state
- Run again
- After n iterations, keep best solution
  - If we have a guess as to the number of optima in the seach space, we can choose n

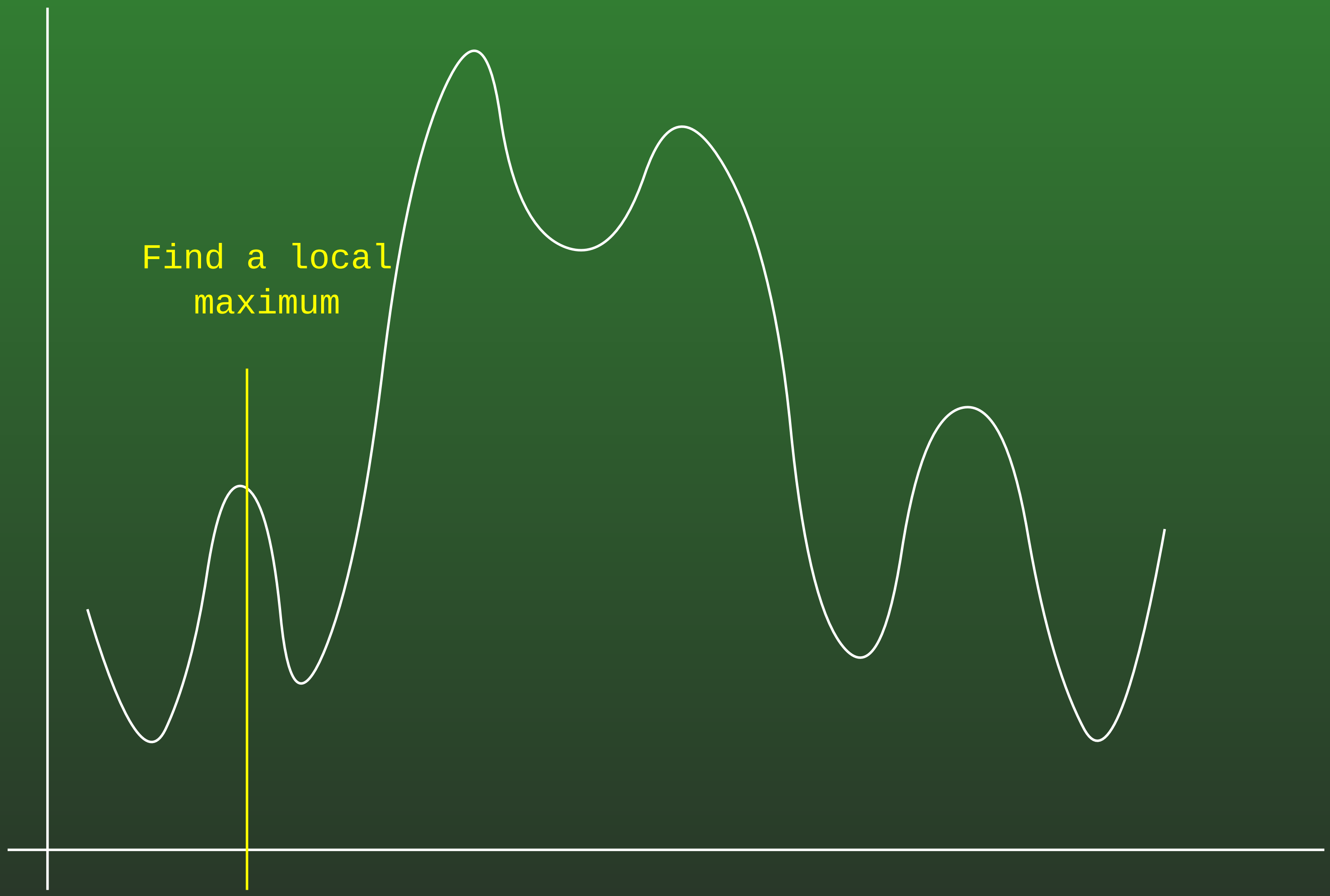
08-13: Simulated Annealing

- Hill Climbing’s weakness: Never moves downhill
  - Can get stuck in local optimum
- Simulated annealing tries to fix this
  - “Bad” (downhill) actions are occasionally chosen to move out of a local optimum

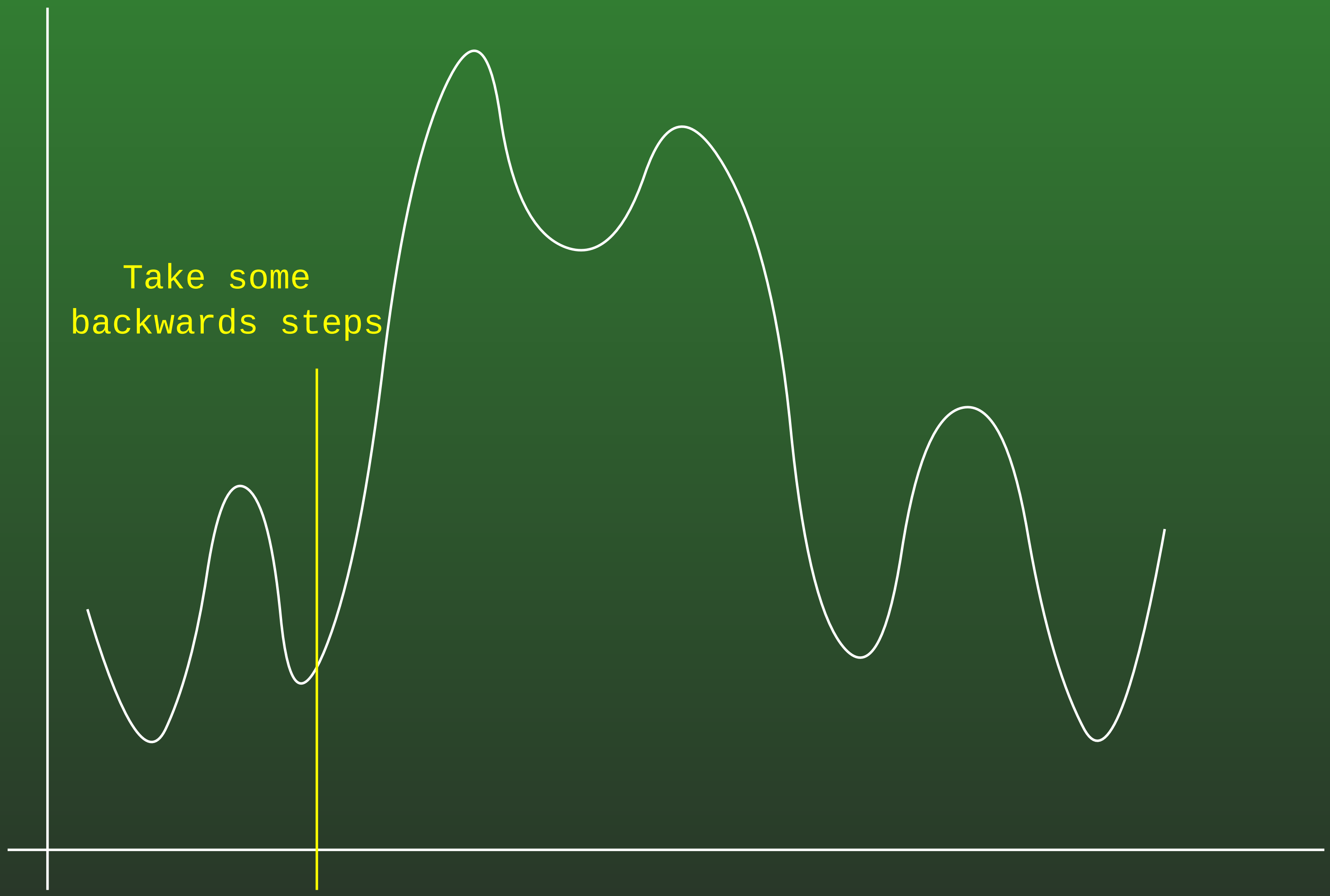
08-14: Simulated Annealing



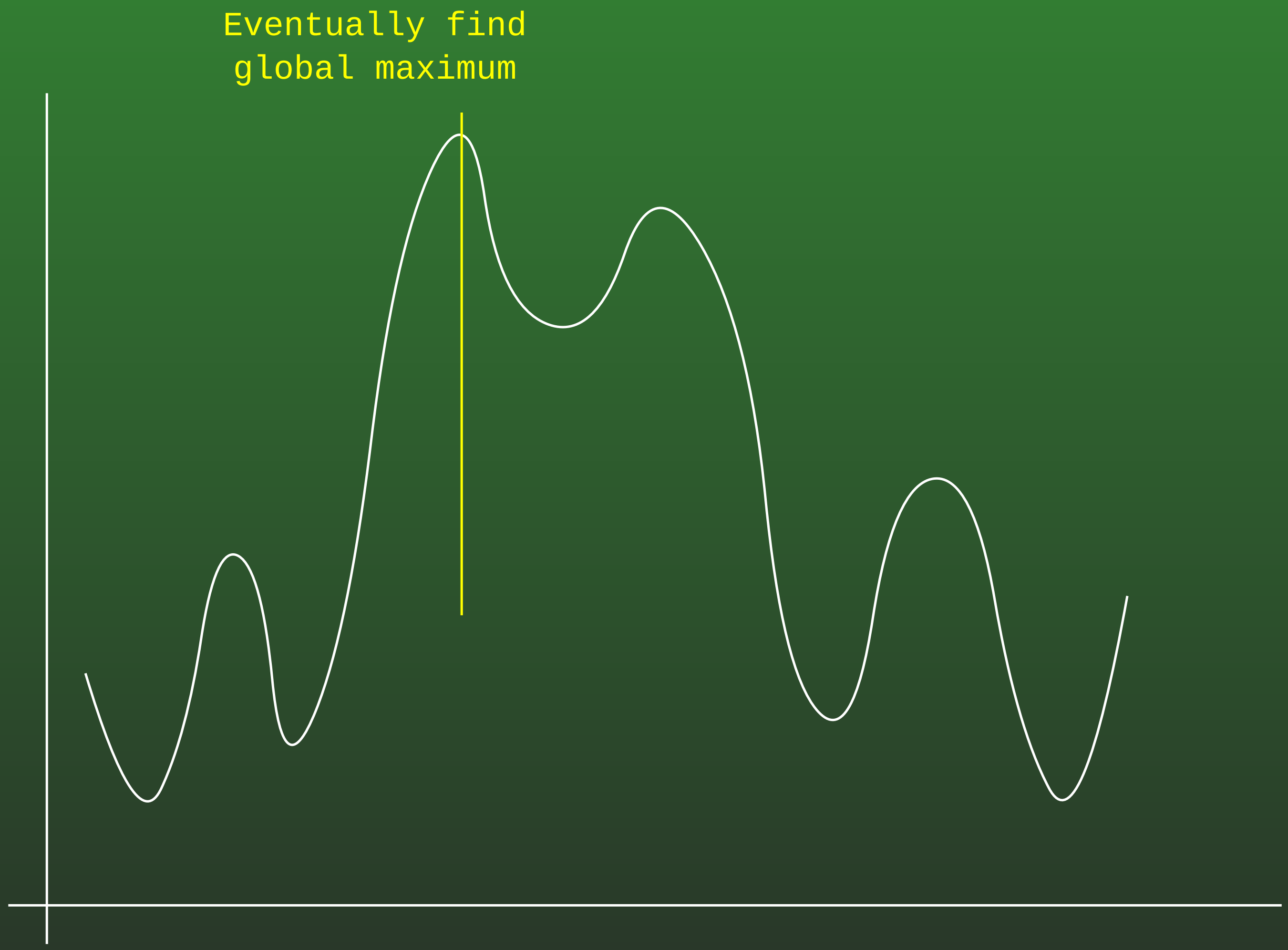
08-15: Simulated Annealing



08-16: Simulated Annealing



08-17: Simulated Annealing



08-18: Simulated Annealing

- Based on analogies to crystal formation
- When a metal cools, lattices form as molecules fit into place
- By reheating and recooling, a harder metal is formed
  - Small undoing leads to a better solution
  - Minimize the “energy” in the system
- Similarly, small steps away from the solution can help hill-climbing escape local optima

08-19: Simulated Annealing

```
T = initial
s = initial-state
while (s != goal)
   ch = successor-fn(s)
   c = select-random-child(ch)
   if c is better than s
      s = c
   else
      s = c with probability p(T,c,s)
   update T
```

- What is T? P?

08-20: Simulated Annealing

- Make “mistakes” (downhill steps) more frequently early in the search, and more rarely later in the search
- T is the “Temperature”
  - High temperature: Make lots of “mistakes”
  - Low temperature: Make fewer mistakes
- P is the probability function, when to make a mistake
- How should T change over time, what should P be?

08-21: Cooling Schedule

- Function for changing T is called a cooling schedule
- Most commonly used schedules:
  - Linear: $T_{new} = T_{old} - dt$
  - Proportional: $T_{new} = c * T_{old}, \quad c < 1$

Boltzmann Distribution

- Probability of accepting a mistake P is governed by a *Boltzmann distribution*
- s is the current state, c is the child being considered, and o is the function to optimize
- $P(c) = exp(\frac{-|o(c)-o(s)|}{T})$
- Boundary conditions:
  - $|o(c) - o(s)| = 0$, then $P(c) = 1$
  - T very high, all fractions near 0, $P(c)$ near 1
  - T low, $P(c)$ depends on $|o(c) - o(s)|$
- Gives us a way of weighing the probability of accepting a “mistake” by its quality

08-23: Boltzmann Distribution

- Simulated Annealing is (theoretically) complete and optimal as long as T is lowered "slowly enough"
  - "Slowly enough" might take more time than exhaustive search
  - Still can be useful for finding a "pretty good" solution
- Can be very effective in domains with many optima
- Simple addition to a hill-climbing algorithm
- Weakness: selecting a good cooling schedule – very hard!
- No problem knowledge used in search (outside of picking cooling schedule)

08-24: Genetic Algorithms

- Genetic Algorithms: “Parallel hill-climbing search”
- Basic Idea:
  - Select some solutions at random
  - Combine the best parts of the solutions to make new solutions
  - Repeat
- Successors are functions of *two* states, rather than one

08-25: GA Terminology

- Chromosome: A solution or state
- Trait / gene: A parameter or state variable
- Fitness: The “goodness” of a solution
- Population: A set of chromosomes or solutions

08-26: Basic GA

```
pop = makeRandomPopulation
while (not done)
  foreach p in pop
    p.fitness = evaluate(p)
  for i to size(pop) by 2:
    parent1, parent2 = select random solutions from pop
      (using fitness)
    child1, child2 = crossover(parent1, parent2)
    mutate child1, child2
  replace old population with new population
```

08-27: Analogies to Biology

- This is *not* how biological evolution works
- Biological evolution is much more complex
- Biology is a nice metaphor
  - ... but Genetic Algorithms must stand or fail on their own merits

08-28: Encoding a Problem

- Choosing an encoding can be tricky
- Traditionally, GA problems are encoded as bitstrings
- Example: 8 queens. For each column, we use 3 bits to encode the row of the queen = 24 bits
- 100 101 110 000 101 001 010 110 = 4 5 6 0 5 1 2 6
- We begin by generating random bitstrings, then evaluating them according to a *fitness function* (the function to optimize)
  - 8 Queens: number of nonattacking pairs of queens (max = 28)

08-29: Generating New Solutions

- Successor function: Work on two solutions
  - Called *Crossover*
- Pick two solutions p_1 and p_2 to be parents
  - Go into *how* to pick parent solutions in a bit
- Pick a random location on the bitstring (locus)
- Merge the first part of p_1 with the second part of p_2 (and vice versa) to produce two new bitstrings

08-30: Crossover Example

- s1: 100 101 110 000 101 001 010 110 = 4560512
- s2: 011 000 101 110 111 010 110 111 = 1056726
- Pick locus = 9
- s1: (100 101 110) (000 101 001 010 110)
- s2: (011 000 101) (110 111 010 110 111)
- Crossover:
- s3: (100 101 110) (110 111 010 110 111) = 4566726
- s4: (011 000 101) (000 101 001 010 110) = 1050512

08-31: Mutation

- Next, apply mutation
- With probability m (where m is small), randomly flip one bit in the solution
- After generating a new population of the same size as the old poplation, throw out the old population and start again

08-32: What is going on?

- Why does this work?
  - Crossover: recombine pieces of partially successful solutions
  - Genes closer to each other are more likely to stay together in successive generations
    - Encoding is important!
  - Mutation: Inject new solutions into the population
    - If a trait was missing from initial population, crossover cannot generate it without mutation

08-33: Selection

- How do we select parents for reproduction?

08-34: Selection

- How do we select parents for reproduction?
- Use the best n percent?
  - Want to avoid premature convergence
  - No genetic variation
  - Poor solutions can have promising subparts
- Random?
  - No selection pressure

08-35: Roulette Selection

- *Roulette Selection* weights the probability of a chromosome being selected by its relative fitness
- $P(c) = \frac{fitness(c)}{\sum_{crh \in pop} fitness(chr)}$
- Normalizes fitness; total relative fitness will sum to 1
- Can use these as probabilities

08-36: Example

- Maximize $f(x) = x^2$ over range $[0, 31]$
  - Assume integer values of x
- Five bits to encode solution
- Generate random initial population

String	Fitness	Relative Fitness
01101	169	0.144
11000	576	0.492
01000	64	0.055
10011	361	0.309
Total	1170	1.0

08-37: Example

- Select parents with roulette selection
- Choose a locus, and crossover the strings

String	Fitness	Relative Fitness
0110 | 1	169	0.144
1100 | 0	576	0.492
01000	64	0.055
10011	361	0.309
Total	1170	1.0

Children: 01100, 1101

Example

- Select parents with roulette selection
- Choose a locus, and crossover the strings

String	Fitness	Relative Fitness
01101	169	0.144
11 | 000	576	0.492
01000	64	0.055
10 | 011	361	0.309
Total	1170	1.0

Children: 01100, 11011 Children: 01011, 10000

08-39: Example

- Replace old population with new population
- Apply mutation to new population
  - With a small population and low mutation rate, mutations are unlikely
- New Generation:
  - 01100, 11001, 11011, 10000
- Average fitness has increased (293 to 439)
- Maximum fitness has increased (576 to 729)

08-40: What's really going on?

- Subsolutions 11*** anbd ****1 are recombined to produce a beter solution
- Correlation between strings and fitness
  - Having a 1 in the first position is correlated with fitness
  - Unsurprising, considering encoding
- Call a 1 in the first position a building block
- GA's work by recombining smaller building blocks into larger building blocks

08-41: Schemas (Schemata)

- Way to talk about strings that are similar to each other
- Add '*' (don't care) symbol to {0, 1}
- A schema is a template that describes a set of strings using {0, 1, *}
  - 111** matches 11100, 11101, 11110, 11111
  - 0*11* matches 00110, 00111, 01110, 01111
  - 0***1 matches 00001, 00011, 00101, 00111, 01001, 01011, 01101, 01111
- Premise: Schemas are correlated with fitness
- In many encodings, only some bits matter for a solution. Schemas give us a way of describing all important information in a string

08-42: Schemas (Schemata)

- GAs process schemas, rather than strings
- Crossover may or may not damage a schema
  - **11* vs 0***1
- Short, highly fit low-order schema are more likely to survive
  - Order: the number of fixed bits in a schema
    - 1**** - order 1
    - 0*1*1* - order 3
- Building Block Hypothesis: GAs work by combining low-order schemas into higher-order schemas to produce progressively more fit solutions

08-43: Schema Theorem

'Short, low-order, above-average fitness schemata receive exponentially increasing trials in subsequent generations."

08-44: Theory vs. Implementation

- Schema Theorem shows us *why* GAs work
- In practice, implementation details can make a big difference in the effectiveness of a GA
  - Encoding Choices
  - Algorithmic improvements

08-45: Tournament Selection

- Roulette selection is nice, but computationally expensive
  - Every individual must be evaluated
  - Two iterations through the entire population
- Tournament selection is a much less expensive selection mechanism
- For each parent, choose two individuals at random
- Higher fitness gets to reproduce

08-46: Elitism

- Discarding all solutions from a previous generation can slow down a GA
  - Bad draw can destroy progress
  - May want monotonic improvement
- Elitism is the practice of keeping a fraction of the population to carry over without crossover
- Varying the fraction lets you tradde current performance for learning rate

08-47: When to Stop

- Stop whenever the GA finds a “Good Enough” solution
- What if we don’t know what “Good Enough” is?
  - When have we found the best solution to TSP?
- Stop when the population has converged
  - Without mutation, eventually one solution will dominate the population
- After “enough” iterations without improvement

08-48: Encoding

- Hardes part of GAs is determining how to encode problem instances
  - Schema threorem tells us short = good
  - Parameters that are interrelated should be located near each other
- n Queens: Assume that each queen will go in one column
- Problem: Find the right row for each queen
- n rows requires $\log_2 n$ bits
- Length of string $n \log_2 n$

08-49: Encoding Continuous Values

- How could we optimize a real-valued function?
- $f(x) = x^2, x \in Reals[0, 31]$
- Break input space into m chunks
- Each chunk is coded with a binary number
- Called *discretization*

08-50: Permutation Operators

- Some problems can't be represented easily as a bitstring
- Traveling Salesman
  - Encoding as a bitstring will cause problems
  - Crossover will produce invalid solutions
- Encode this as a list of cities: [3, 1, 2, 4, 5]
- Fitness: MAXTOUR - tour length (so we can have a maximization problem, rather than a minimization problem

08-51: Partially Matched Crossover

- How to do crossover?
- Exchange *positions* rather than substrings
- Example:
  - t1: 3 5 4 6 1 2 8 7
  - t2: 1 3 6 5 8 7 2 4
- First, pick two loci at rancom

08-52: Partially Matched Crossover

- t1: 3 5 | 4 6 1 2 | 8 7
- t2: 1 3 | 6 5 8 7 | 2 4
- Use pairwise matching to exchenge corresponding cities on each tour
  - In each string, 4 and 6 trade places, as do 6 and 5, 1 and 8, and 2 and 7
  - New children
    - c1: 3 6 5 4 8 7 1 2
    - c2: 8 3 4 6 1 2 7 5
- Intuition: Building blocks that are sections of a tour should tend to remain together

08-53: Partially Matched Crossover

- Partially Matched Crossover is one of many approaches to using GAs to solve permutation problems
- Could also encode the position of each city
- Can replace subtours

08-54: Summary

- Local search
  - Looking for a state, not a path
  - Just store the current state
  - Easy to code, low memory – problems?
- Simulated Annealing
  - Finding appropriate cooling schedule difficult
  - Theoretically complete, in practice useful when lots of acceptable solutions

08-55: Summary

- Genetic Algorithms
  - Use bitstrings to perform local searches through a space of possible schemas
  - Lots of parameters to play with in practice
  - Representation is hardest part of problem
  - Effective at searching vast spaces
  - Sensitive to parameters
    - Mutation Rate
    - Elitism Rate
    - Initial Population