Lecture 7

CS 6380 Artificial Intelligence

Local Search

Department of Computer Science and Engineering

Indian Institute of Technology Madras (IITM), India

Some material based on lecture slides for CSE 571: Artificial Intelligence by Siddharth Srivastava, ASU.

Logistics and Recap

Talk Announcement

- **Title:** Towards Reliable Autonomy: From Individual Agents to Autonomous Teams
- **Date:** August 21, 2026
- **Time: 11**:00 AM – 12:00 PM
- Location: SSB 334

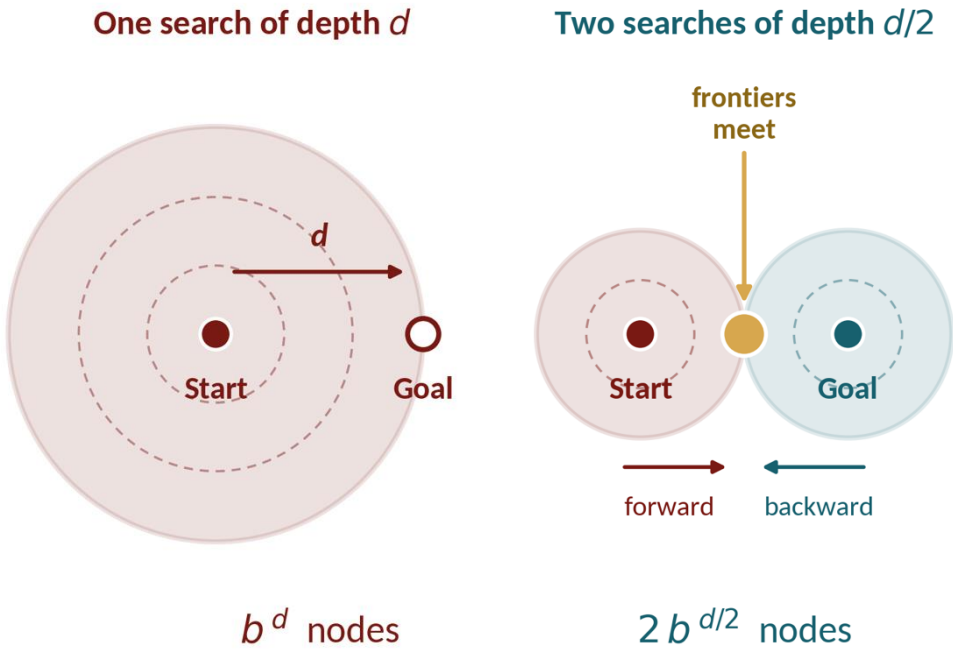
Dr. Sandhya Saisubramanian
Oregon State University, USA
https://www.sandhyasai.com/

Methodology for Generating Heuristics

- Create a relaxed version of the problem
  - Abstract, or approximate and easier
- Relaxed problems allow shortcuts
  - Optimal solution cost in relaxed $\leq$ optimal solution cost in real
- Optimal solution cost in the relaxed problem constitutes a heuristic
- But, if the relaxed problem is not easy, computing its optimal can be a challenge!

Bidirectional Search

- Idea: run two searches at once:
  - forward from the start,
  - backward from the goal, and
  - stop as soon as the two frontiers intersect.
- The path is the forward path to the meeting node, joined to the reversed backward path.
- Each search only has to reach depth $d/2$. Because node counts grow exponentially with depth, halving the depth takes a square root of the work.



What it costs you: an explicit goal *state* to search back from, a goal *test* alone is not enough, and a way to generate predecessors, which means the actions must be reversible.

Bidirectional Search: The Payoff and the Catch

A concrete case: branching factor b = 10, solution depth d = 10.

10,000,000,000

b^d **nodes**

one search of depth 10

200,000

$2b^{d/2}$ **nodes**

two searches of depth 5

50,000×

fewer nodes

generated to find the same path

You still store a frontier

Every generated node must be tested against the other frontier, so at least one of them has to be held in memory: space stays $O(b^{d/2})$.

Predecessors must exist

Easy on a road map, where every road runs both ways. Hard when an action destroys information and cannot be undone.

Goals given as a test

"Checkmate" describes millions of states, not one. With no single state to search back from, the backward search has nowhere to start.

First meeting ≠ cheapest

With non-uniform costs the first frontier intersection need not lie on an optimal path, so bidirectional UCS cannot simply stop there.

More Videos of Search Algorithms

https://movingai.com/SAS/

Let's Search Again: Local Search Algorithms

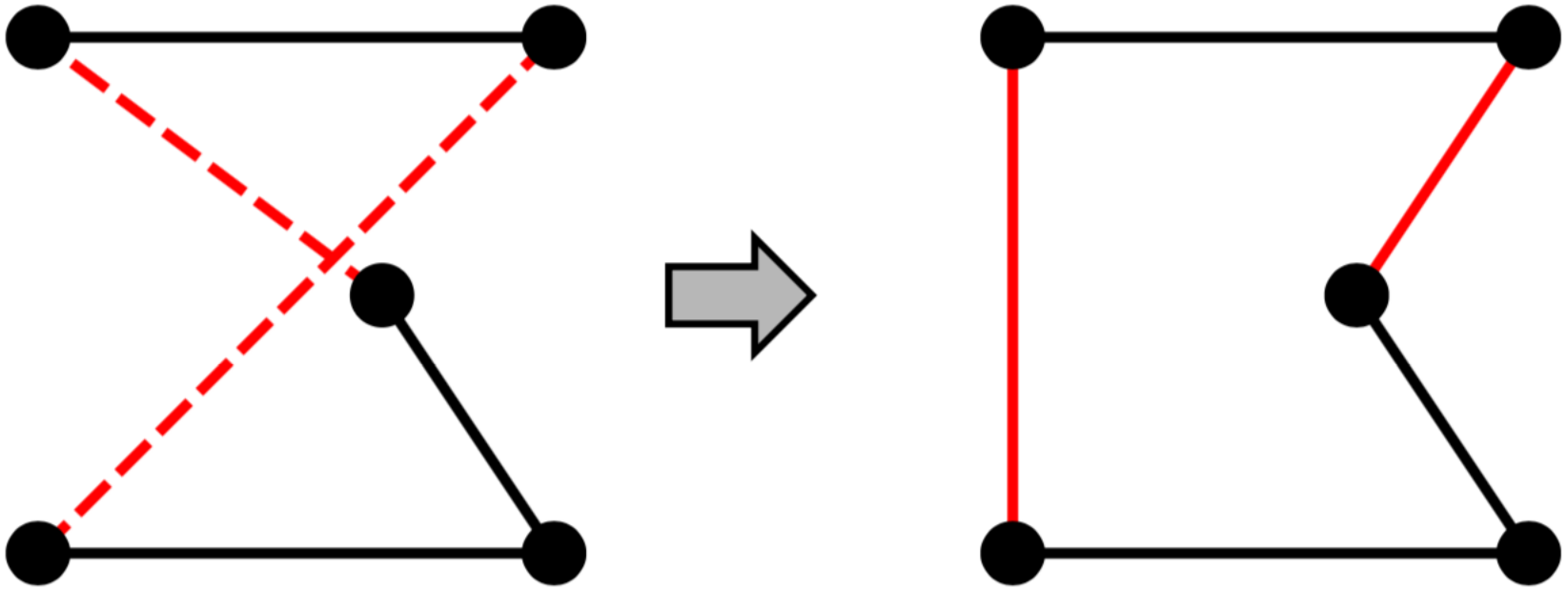
Simple tricks that seem to do well!

Local Search and Iterative Improvement

- In some search problems, finding a goal state is important
  - E.g., classroom scheduling
- Path used to find it is not included in the performance measure

- Iterative improvement algorithms for search (or "local search" algorithms) can be used for such problems
- Idea:
  - Maintain a "current state"
  - Try to improve it
  - That's it! No closed list, fringe management, node data structures
  - Constant space!
  - Time?

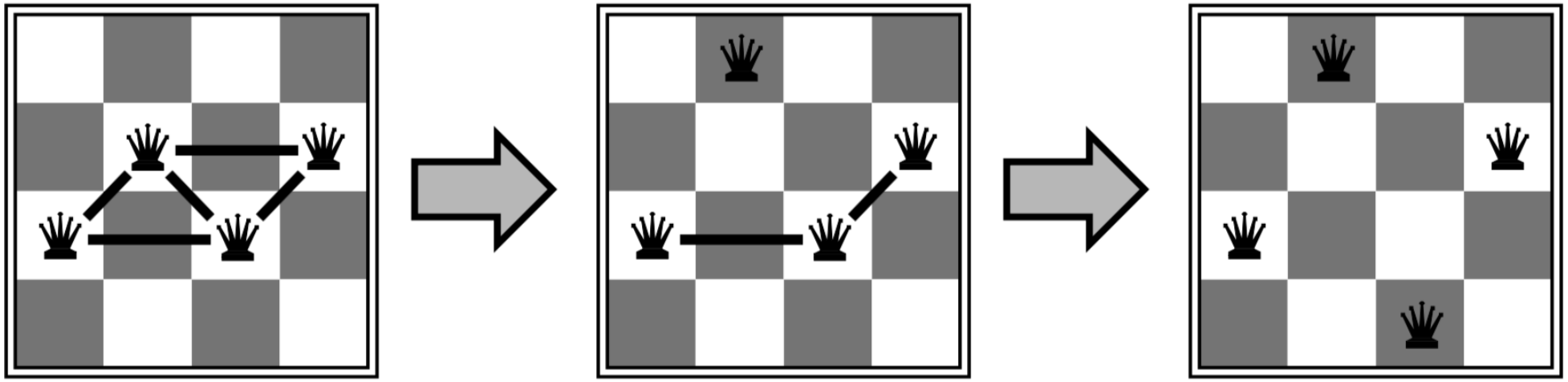
Local Search for Traveling Salesperson Problem



- Start with any complete tour, do pairwise swaps of edge endpoints
- Strange but true: variants of this approach get within 1% of optimal with thousands of cities

Local Search for n-queens

- Place n queens on an $n \times n$ chessboard
  - No queen should be able to attack another
- Local search idea: move any one queen to reduce conflicts



- Strange but true: solves n queens instantaneously for very large n (~1 million)

Hill-Climbing or Gradient Descent/Ascent

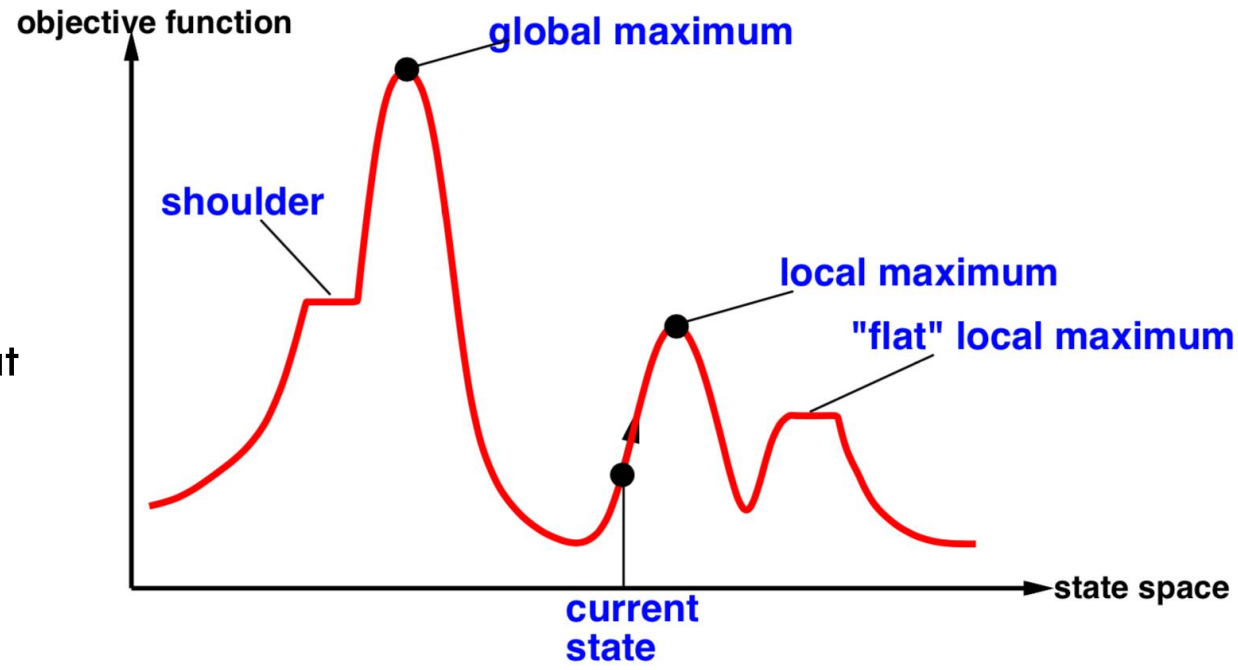
```
function HILL-CLIMBING( problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    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
```

"Like climbing Everest in thick fog with amnesia"

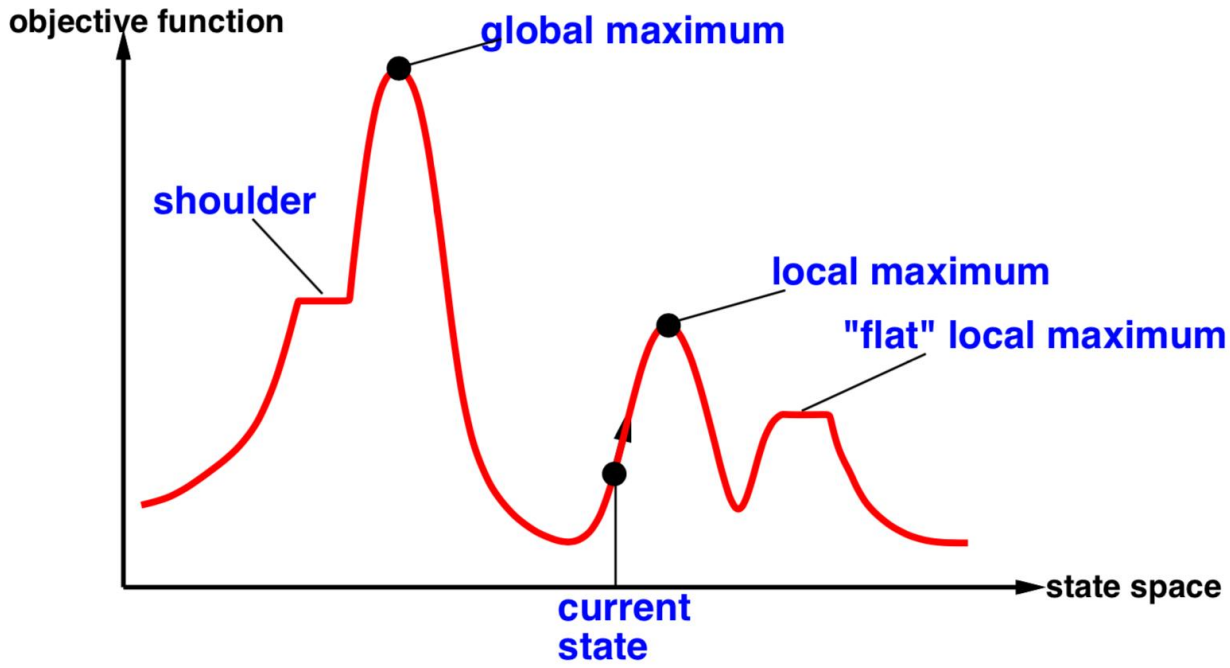
Problem with HC: local optima!

- Trivial fixes:
  - Randomized restarts
    - Problem: no bound on trials
  - Random sideways moves
    - Escapes from most local optima, but not flat regions
- What can we do to jump out of local optima?



The Landscape: Where Hill Climbing Stops

- Picture the search as a landscape: position = state, height = value
- Local maximum: no neighbour is better, but this is not the best peak
- Plateau: a flat region, every neighbour scores the same
  - a shoulder is flat but has a way up off its far edge
  - a flat local maximum has none
- Hill climbing is greedy and myopic: it only ever compares a state against its immediate neighbours



The same picture upside-down is the cost landscape — then we descend.

Hill-Climbing Variants

Same skeleton: the only thing that changes is which uphill move you take, and what you do when there isn't one.

Steepest ascent

Evaluate every neighbour, move to the best one. Expensive when states have many neighbours, 8-queens has $8 \times 7 = 56$ of them, and it is the variant that plateaux stop dead.

Stochastic hill climbing

Choose at random among the uphill moves, with probability weighted by how steep each is. Slower per solution, but it explores more of the landscape and often ends up somewhere better.

First-choice hill climbing

Generate successors at random and take the first one that improves. The right choice when a state has thousands of neighbours and you cannot afford to score them all.

Random-restart hill climbing

When you get stuck, throw the state away and start again from a random one. Trivially complete, sooner or later a restart lands on a goal. “If at first you don't succeed, try, try again.”

Does Any of This Work? 8-Queens

Steepest-ascent hill climbing on 8-queens, starting from a random state each time.

~17 million
states to search
8^8 complete configurations

14%
of runs reach a solution
the other 86% get stuck

94%
if sideways moves are allowed
capped at 100 in a row

It succeeds or fails quickly

About 4 steps to reach a solution, 3 to get stuck. Failure is so cheap that restarting is a perfectly sensible strategy. With sideways moves the runs lengthen to roughly 21 and 64 steps.

Restarts: do the arithmetic

With success probability p, the expected number of restarts is $1/p \approx 7$, so about 22 steps in total. A 17-million-state problem, solved in well under a second.

Why it works here

This landscape has few local maxima relative to its size, and they are shallow. That is a property of the problem and the encoding; not evidence that hill climbing is good in general.

How far it scales

Local search solves n-queens instances with millions of queens in about a minute. No systematic search algorithm from Chapter 3 comes anywhere close on these problems.

Simulated Annealing

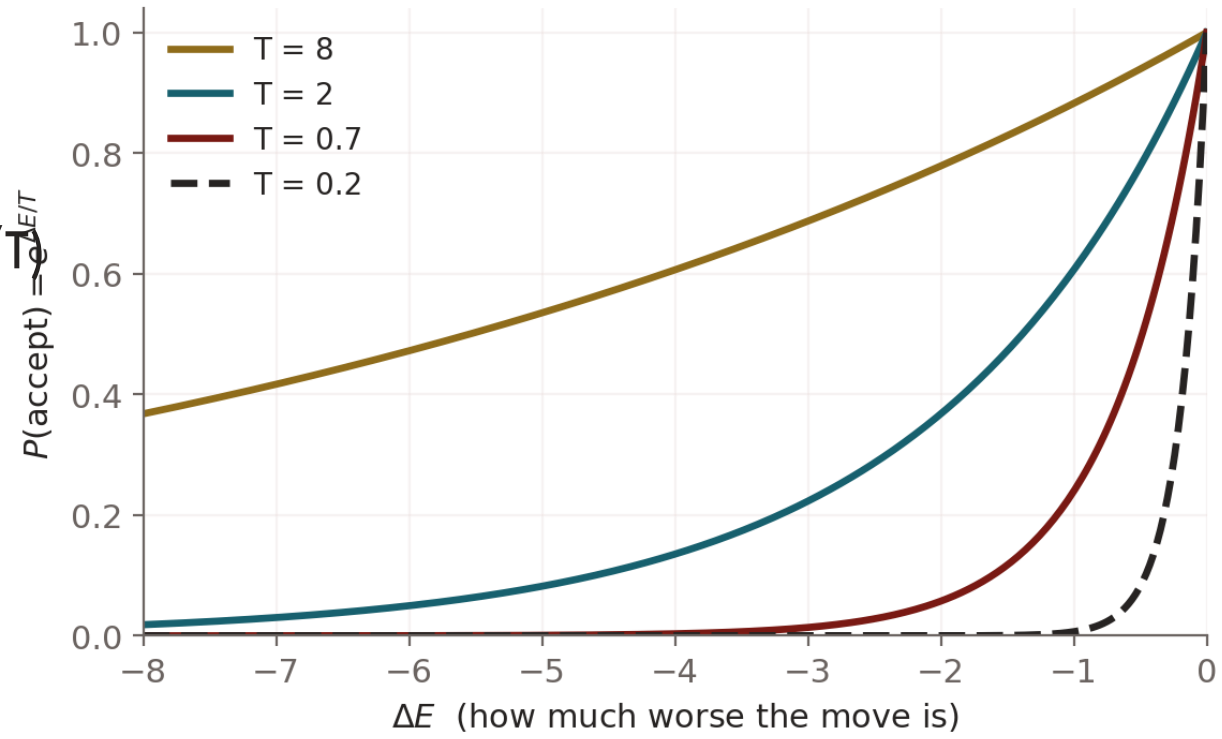
- Allow some bad moves to points far from current value (local optima)
- But decrease their tolerance, frequency over time

```
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)
```

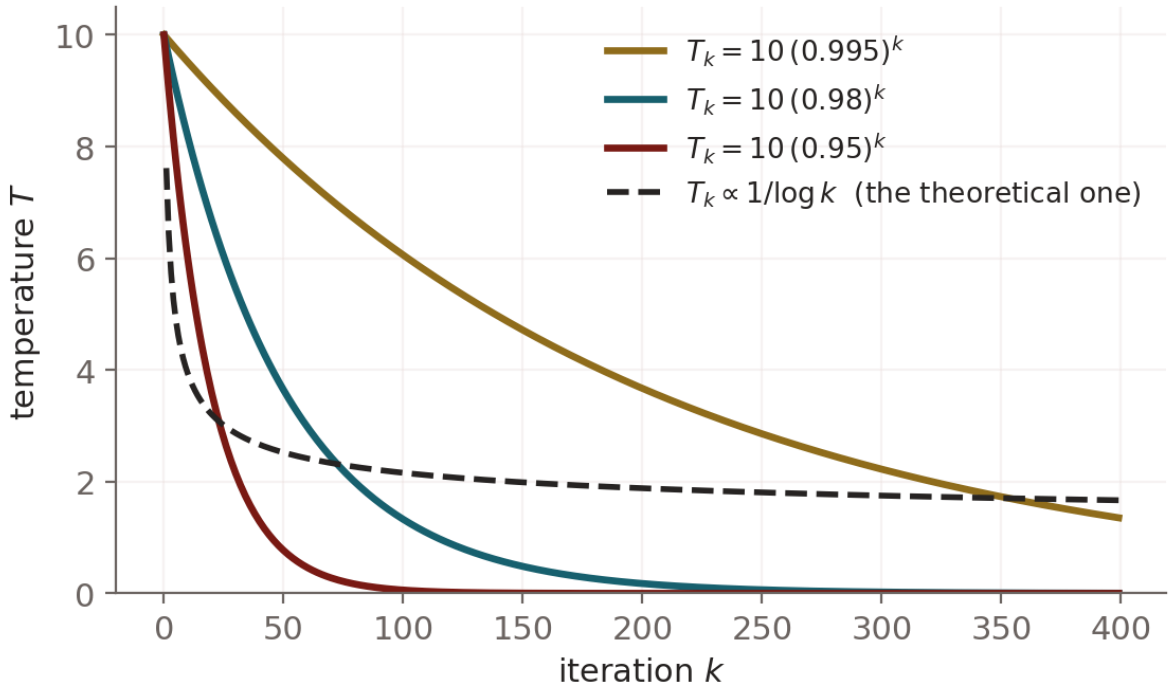
Accepting Bad Moves: the Rule

- Pick a successor at random (not the best one)
- ΔE = value(next) − value(current)
- If ΔE > 0, take it: uphill is always accepted
- If ΔE ≤ 0, take it anyway with probability e^(ΔE/T)
- Two dials control everything:
  - how bad the move is
    - big drops are exponentially rarer
  - the temperature T
    - at high T almost anything is accepted
- *A bad move is never impossible, just improbable*



The Temperature Schedule

- T large → nearly every move accepted → a random walk
- T → 0 → no bad move accepted → ordinary hill climbing
- The schedule interpolates between the two: explore first, exploit later
- In practice, a geometric schedule: $T_k = T_0\,\alpha^k$, with α between 0.8 and 0.99
  - choose T_0 so that roughly half of the bad moves are accepted at the start
  - stop when the acceptance rate collapses, then hill-climb to finish
- *Reheating, restarts and a good neighbour function matter far more than the exact curve*



Simulated Annealing: Theory vs. Practice

What the theorem says

If the temperature is lowered slowly enough, the search converges to a global optimum with probability approaching 1.

At a fixed T the search settles into the Boltzmann distribution, $P(x) \propto e^{(E(x)/T)}$: it spends exponentially more time in good states than in bad ones.

What that is worth

"Slowly enough" can mean exponentially slowly, the schedule the proof needs decays like $1/\log k$.

And random-restart hill climbing also finds the optimum in the limit, for a much sillier reason. A guarantee you can never afford to collect is not a reason to pick an algorithm.

Why it is used anyway

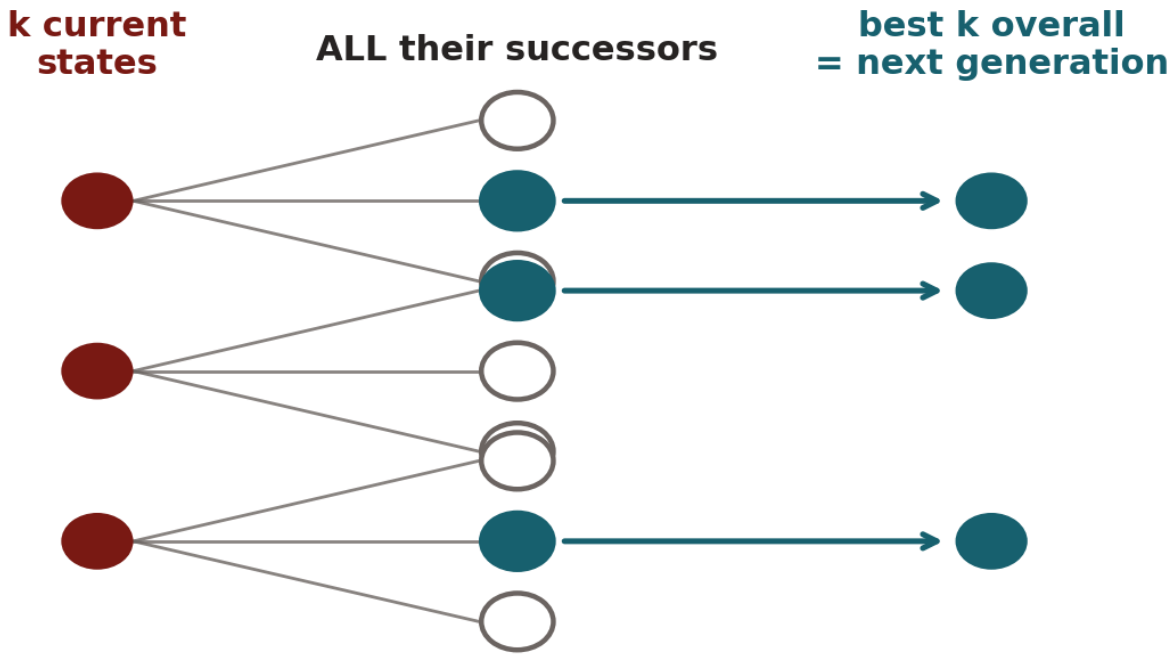
It is a dozen lines of code, needs no gradient, and copes with rugged landscapes that defeat greedy methods.

VLSI layout and placement, factory scheduling, protein folding, large TSP instances: annealing and its relatives are a standard workhorse.

The useful takeaway: judge a local search algorithm by how it performs on your landscape within your compute budget, not by what it converges to in the limit.

Local Beam Search

- Keep k states at once, not one
- Each round:
  - generate ALL the successors of all k states
  - if any is a goal, stop
  - otherwise keep the best k of the whole pool and repeat
- Memory stays O(k): still constant, still no path
- Start the k states randomly, as with restarts



the k threads share one pool — a good region attracts them all

Beam Search Is Not k Restarts

k random restarts

k independent runs. Each one climbs its own hill and learns nothing from the others. The only thing shared is that you take the best answer at the end.

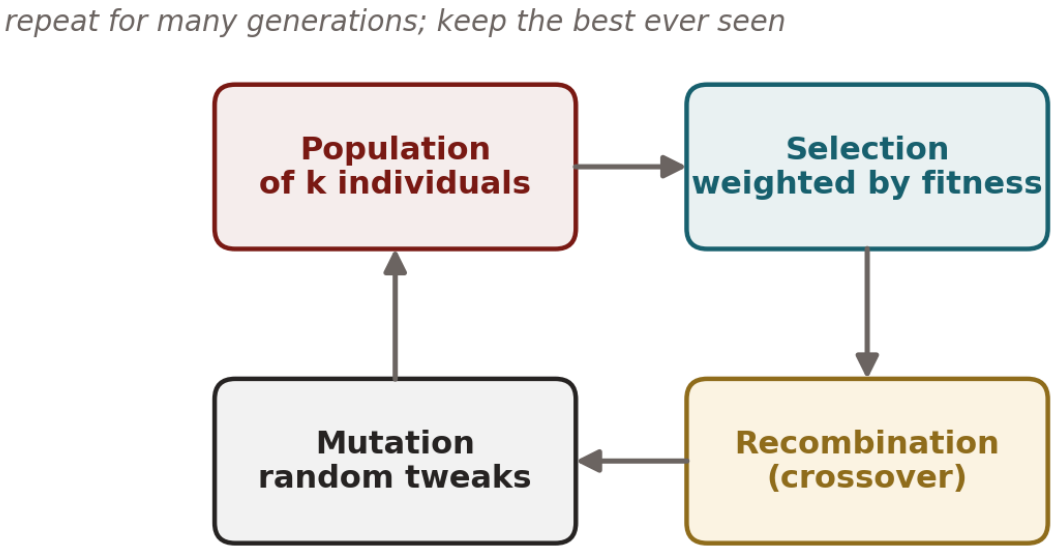
Local beam search

One pool. Selection happens across all k threads at once, so a thread that finds a good region takes slots away from the ones that haven't. “Come over here, the grass is greener!”

- The failure mode: all k states crowd into one small region, and you are paying k times over for a single hill climb
- Stochastic beam search fixes it: instead of taking the best k, draw k successors at random with probability rising with value
  - keeps diversity in the population while still favouring good states
  - and it is exactly natural selection: successors are offspring, value is fitness, and the population size is fixed

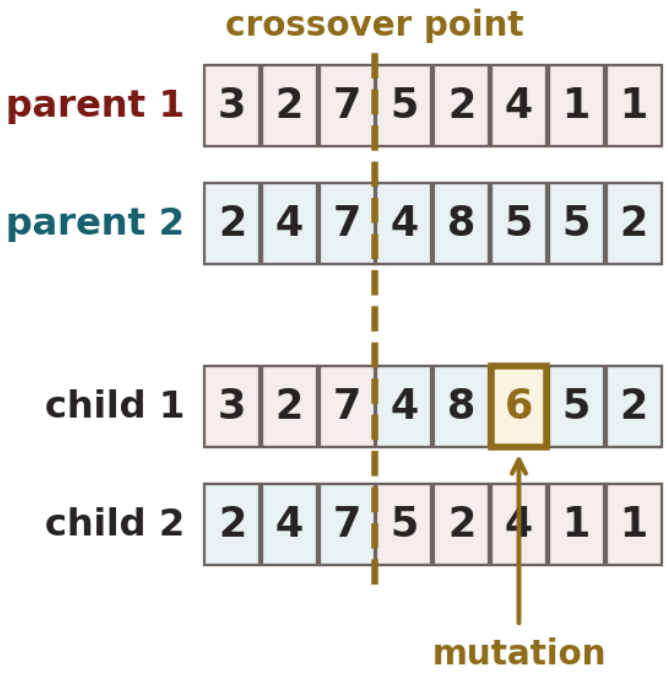
Genetic Algorithms

- Stochastic beam search + a second parent: successors are made by combining two states, not by mutating one
- The ingredients:
  - a population of k individuals, each a string over a finite alphabet
  - a fitness function scoring each individual: higher is better
  - selection, recombination (crossover), and mutation
- Run for many generations; return the best individual ever seen
- *The claim being made: good solutions have good parts, and parts can be recombined*



A Genetic Algorithm on 8-Queens

- Encoding: eight digits, one per column, giving the row of that column's queen: so every string is a legal state
- Fitness: the number of non-attacking pairs of queens; 28 means solved
- Selection: sample parents with probability proportional to fitness: a 23-fitness string is picked more often than an 11



state = one digit per column = the row of that column's queen

fitness = number of non-attacking pairs (28 = solved)

The Design Choices That Actually Matter

Population size k

Too small and diversity dies in a few generations; too large and you waste evaluations. Usually tens to thousands.

Representation

The encoding decides whether crossover means anything. Bit strings, digit strings, real vectors, program trees.

Selection scheme

Fitness-proportional, tournament (pick 2, keep the better), or rank-based. Rank and tournament resist a single superfit individual taking over.

Recombination

One-point, two-point or uniform crossover. The mixing number ρ is how many parents; ρ = 1 is exactly stochastic beam search.

Mutation rate

The only source of genuinely new material once the population converges. Too high and it becomes a random walk.

Elitism and culling

Copy the best individuals forward unchanged so you never lose ground; drop everything below a fitness threshold before selecting.

A genetic algorithm is not one algorithm: it is a template with six knobs, and the results you get are mostly about how you set them.

Does Crossover Actually Help?

- Crossover helps only when the encoding puts related choices next to each other
- A schema is a template such as 246*****: all states whose first three queens sit in rows 2, 4, 6
  - if the instances of a schema score above average, its share of the population grows
  - so a good encoding lets useful blocks survive being cut, and spread
- The digit-per-column encoding has that locality. A random re-ordering of the same bits would not: same states, same fitness, far worse algorithm
- The honest position: on most problems a well-tuned hill climber with restarts matches a genetic algorithm. The biological metaphor is a source of ideas, not evidence
- Relatives worth knowing: evolution strategies (real-valued, self-adapting mutation size), genetic programming (individuals are programs), neuroevolution