Chapter 4, Section 4.1

Local Search Algorithms

Hill-Climbing, Simulated Annealing, Genetic Algorithms

COMP338 - Artificial Intelligence

Birzeit University | Second Semester 2025/2026

Reference: Russell & Norvig, AIMA 4th Edition (2021), Section 4.1

Imagine you are already on the mountain

Sometimes the question is not how to get somewhere. It is how to do better from where you are.

A small story

You are hiking in thick fog. Your goal is the highest point on the mountain. You cannot see the summit.

What can you do? Feel the ground around you. Step toward the steepest rise. Repeat. Stop when no step rises anymore.

You may not reach the highest summit overall, but you will at least reach a peak nearby. Often that is good enough.

The local search principle

1. You hold one **current state**.
2. You can evaluate its **neighbours** (states reachable in one small step).
3. You move to a neighbour that is **better**.
4. You repeat until no neighbour improves things.

No map. No memory of where you came from. No plan.
Just: improve from here.

Why local search? A different kind of problem

Classical search finds a path. Local search finds a good state. The path does not matter.

Classical search (what we did)

Question: what sequence of actions leads from start to goal?

We care about: the path itself.

Examples:
- Romania route from Arad to Bucharest
- Sliding tile sequence in 8-puzzle
- Web crawler navigation

Stores frontier, reached set, parent pointers.

Local search (today)

Question: what state has the best value?

We care about: the destination state, **not how** we got there.

Examples:
- Place n queens with zero conflicts
- Shortest TSP tour through n cities
- VLSI chip layout, airline scheduling
- Neural network weight tuning

Stores just a current state. Very little memory.

Where classical search fails

Three real problems. Each has so many states that systematic enumeration is impossible.

n-queens

Place n queens on an n by n board so no two attack each other. Used in constraint satisfaction.

Brute force at n=8:

C(64, 8) = 4.4 billion configurations

Classical search would enumerate placements. Even pruned, the tree is enormous. Local search reaches a solution in milliseconds.

Travelling salesperson

Visit n cities in a cycle minimizing total distance. Used in delivery routing.

Tours at n=50 cities:

$(50-1)! / 2 \approx 3 \times 10^{62}$

More tours than atoms in the observable universe. No search algorithm can enumerate them. UPS, FedEx, Amazon use local search variants daily.

VLSI chip layout

Place transistors and route wires on silicon. Used in computer chip design.

Modern chips:

Billions of transistors to place

The problem that motivated simulated annealing in 1983 (Kirkpatrick, IBM Research). Still solved by local search today.

What is local search?

Definition

Keep a single **current state**. Try to improve it by moving to a neighbour. Repeat until no neighbour is better, or until time runs out.

Each state has a numerical **objective function** (maximize) or **cost function** (minimize). Neighbours are reached by small changes to the current state.

Tiny memory

Holds only the current state (or a small population). No frontier, no reached set. Memory is O(1) in problem size.

Handles huge spaces

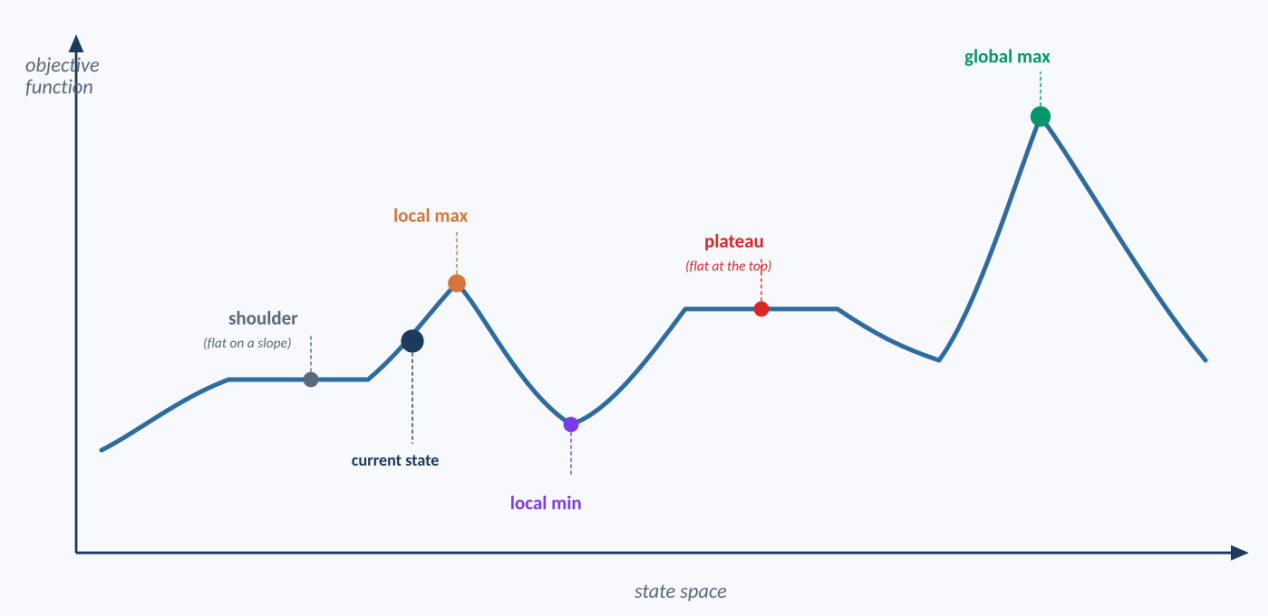
Works on problems where systematic search (BFS, A*) cannot enumerate the state space at all.

Typically incomplete

Can get stuck in suboptimal solutions. Often returns a good answer, not necessarily the best. The trade-off for scalability.

State-space landscape

Imagine the states laid out on a horizontal axis. Each state's height is its objective value.



Five features

Shoulder

flat region on a slope

Local max

peak that is not the best

Local min

valley between peaks

Plateau

flat region at the top

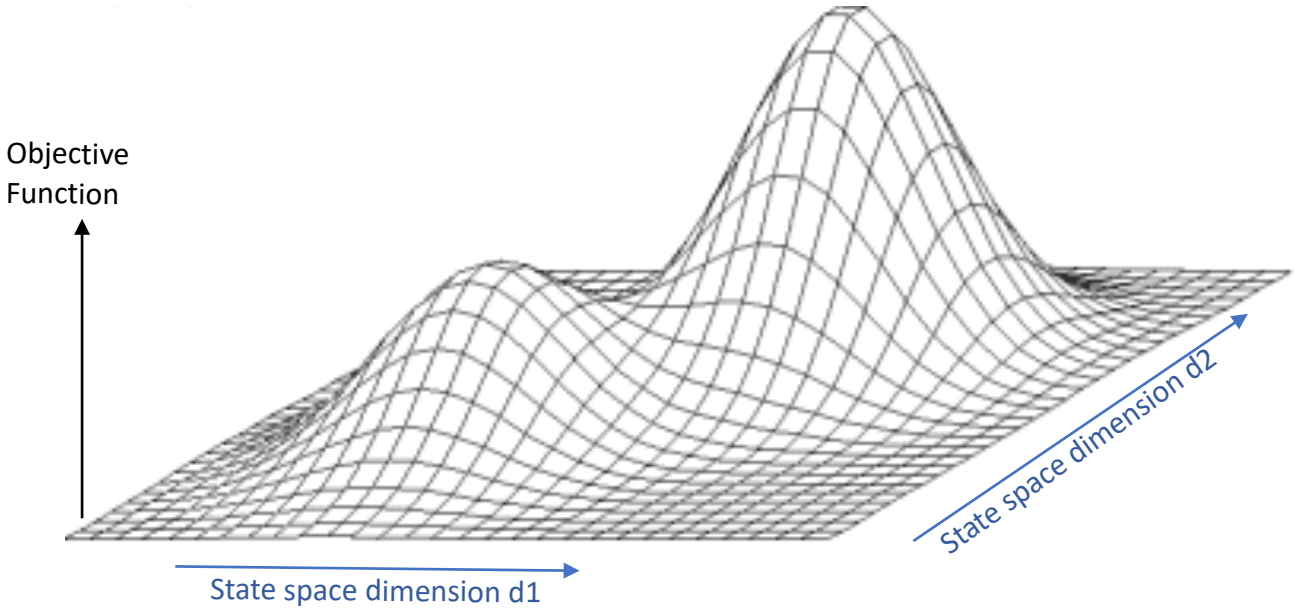
Global max

the truly best state

Hill climbing is myopic. *From the current state (navy), the only neighbours visible are immediately to the left and right. The algorithm climbs to the local max and stops, missing the global max.*

Search landscape in three dimensions

Real problems have many dimensions. Two are easy to draw. The principles are the same.



In higher dimensions

More neighbours per state. For 8-queens, 56 per state. For TSP-50, over 1,000.

More local maxima. More directions in which to be locally optimal.

Cannot be visualized. We trust the algorithm and the math.

Same concepts apply. *Peaks, valleys, plateaus, shoulders, ridges.*

Today's lecture: outline

1. **Hill-climbing search**

   The simplest local search. Greedy. Stops at any peak. Worked on 8-queens.

2. **Simulated annealing**

   Accept worse moves with probability. Escapes local maxima. Converges to global max in the limit.

3. **Genetic algorithms**

   Population of candidate solutions. Selection, crossover, mutation. Inspired by evolution.

4. **Comparison and wrap-up**

   When to use which algorithm.

01

Hill-climbing search

Continually move uphill. Stop at a peak. Simple, fast, gets stuck.

Hill-climbing: the idea

Core principle

From the current state, look at all neighbours. Pick the one with the highest objective value. Move there. Repeat. Stop when no neighbour is better.

Also called **greedy local search** - always takes the locally best step, never looks back or sideways.

1	**Evaluate current**	Compute the objective value of the current state.
2	**Generate neighbours**	Apply the neighbour function to enumerate one-step modifications.
3	**Pick the best**	Find the neighbour with highest objective value.
4	**Compare and decide**	If the best neighbour is better than current, move there. Otherwise stop and return current.

Hill-climbing: annotated pseudocode (AIMA Fig. 4.5)

```
function HILL-CLIMBING(problem):
    current  <-  problem.INITIAL_STATE

    loop:
        neighbour  <-  successor of current
                       with highest VALUE

        if VALUE(neighbour)  <=  VALUE(current):
            return current

        current  <-  neighbour
```

VALUE returns the objective function. For costs, replace VALUE with -COST and the same logic minimizes.

Initialize
Start with the problem's initial state. No special setup - just one state in memory.

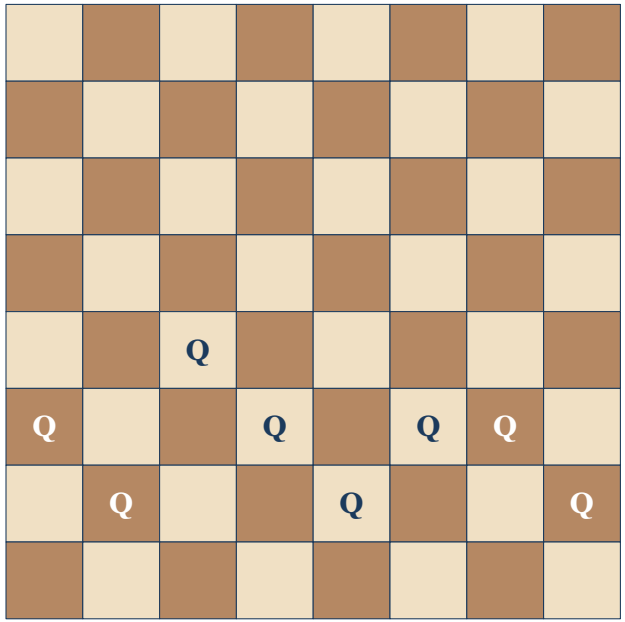
Find the best neighbour
Enumerate all one-step neighbours and pick the one with highest objective value. This is the steepest-ascent variant. Tie-break randomly if needed.

Stop if no improvement
If even the best neighbour is no better than current, we have reached a peak (or plateau or local max). Return current as the answer.

Move uphill
Otherwise, commit to the best neighbour. Discard current. No memory of past states.

Example: 8-queens with hill-climbing

Each column holds one queen. The goal: no two queens attack each other.



h = 17

(17 pairs of queens attack each other)

Local search formulation

State: an 8-vector. Each position c records the row (0-7) of the queen in column c. The shown state is `[5, 6, 4, 5, 6, 5, 5, 6]`.

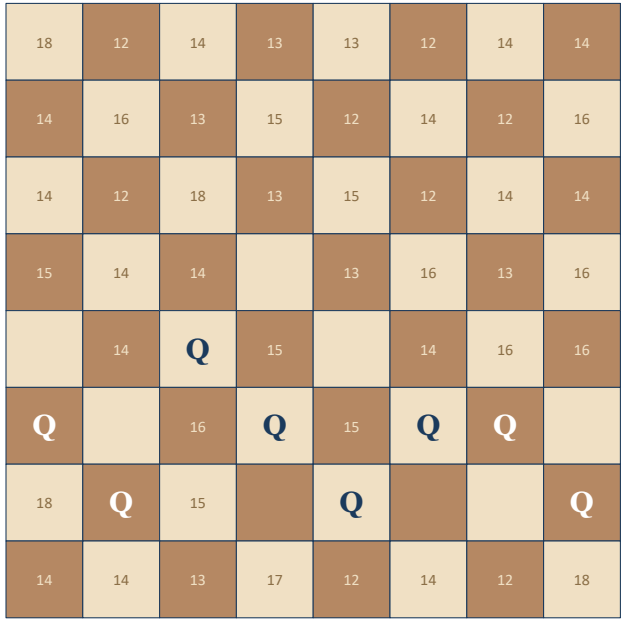
Neighbour: move any single queen up or down within its column. Each state has 8 * 7 = **56 neighbours**.

Cost function h: number of pairs of queens attacking each other directly or indirectly (same row, same diagonal). **Goal: h = 0**.

Hill climbing here = *move the queen whose move most reduces h.*

Picking the best move: successor h-values

Each empty cell shows what h would be if we moved that column's queen to that row. We pick the smallest.



Picking the steepest descent

Current h = 17.
Smallest neighbour h shown on the board: 12.

Best move: any cell with h = 12. Move that column's queen there.

There are ties. Multiple cells show h = 12. Steepest-ascent tie-breaks at random.

After the move, h drops to 12. New state. Repeat.

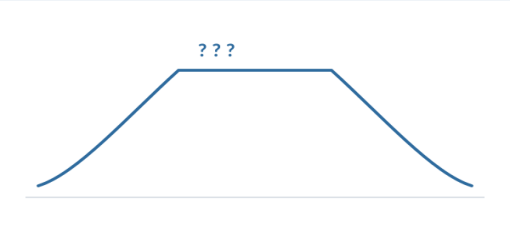
Where hill-climbing fails

Three landscape features that defeat naive hill climbing.

Local maximum

All neighbours are worse but we are not at the global max. Hill climbing stops here, satisfied. For 8-queens, AIMA reports 86% of random starts end at a local min with h > 0.

Plateau or shoulder



Neighbours have equal value. Hill climbing cannot decide where to go. The flat region may extend for many steps. Steepest-ascent stops; sideways moves help but can loop.

Ridge

Sequence of local maxima not directly connected. To move from one ridge peak to the next, hill climbing must briefly descend, which it refuses to do. Like trying to walk along the top of a steep cliff.

Hill-climbing variants

Beyond Jarrar's slides

Refinements that mitigate the failure modes - each fixes a different problem.

Stochastic hill climbing

Choose randomly among uphill neighbours, weighted by steepness. Slower per-step than steepest-ascent but often finds better answers. Helps escape ridges by varying the direction of climb.

First-choice hill climbing

Generate neighbours one at a time in random order. Take the first one that improves. Useful when each state has thousands of neighbours and evaluating all is too expensive (think: TSP with many cities).

Random-restart hill climbing

Run plain hill climbing from many random starting states. Return the best solution found across all runs. Probabilistically complete: with enough restarts, eventually one will land in the basin of the global max.

Hill climbing with sideways moves

On a plateau, allow lateral moves up to a bounded number of times. Helps cross shoulders. Risk: can loop indefinitely on flat local maxima.

Hill-climbing: properties

Complete

No (without restart). Stops at any local max.

Plain hill climbing returns whatever peak it reaches. With random-restart, becomes probabilistically complete - with enough restarts, eventually lands in the global basin.

Optimal

No. Returns local max, not necessarily global.

Even random-restart is only probabilistically optimal: depends on chance of starting in the right basin. For 8-queens, 86% of starts hit local minima.

Time

Per step: O(b) to evaluate neighbours. Total: depends on starting state.

Each iteration scans all b neighbours. Number of iterations until peak varies. For 8-queens: ~4 steps on success, ~3 on failure. With restarts, multiply by expected number of restarts.

Space

O(1) - one current state.

No frontier, no reached set, no history. Constant memory regardless of problem size. The defining advantage of local search.

02

Simulated annealing

Escape local maxima by occasionally taking worse moves. Probability of acceptance falls over time.

Simulated annealing: the intuition

The metallurgy analogy (Kirkpatrick, Gelatt, Vecchi 1983)

Heat metal to high temperature, then cool slowly. At high temperatures, atoms vibrate energetically and can rearrange freely. As temperature drops, motion settles into a low-energy crystalline structure - a near-global minimum of the energy function.

Apply this to search: start by accepting many bad moves, gradually restrict to only good ones.

Hill climbing's problem

Never accepts a worse move. So once trapped at a local maximum, never escapes.

Picture: hiker on a small hill, with a bigger mountain across a valley. To reach the mountain, must first walk downhill. Hill climbing refuses.

SA's fix

Pick a random neighbour. If better, take it. If worse, take it **with a probability** that depends on how much worse, and how 'hot' we are.

Early (hot): many bad moves accepted. The hiker can climb out of small valleys.

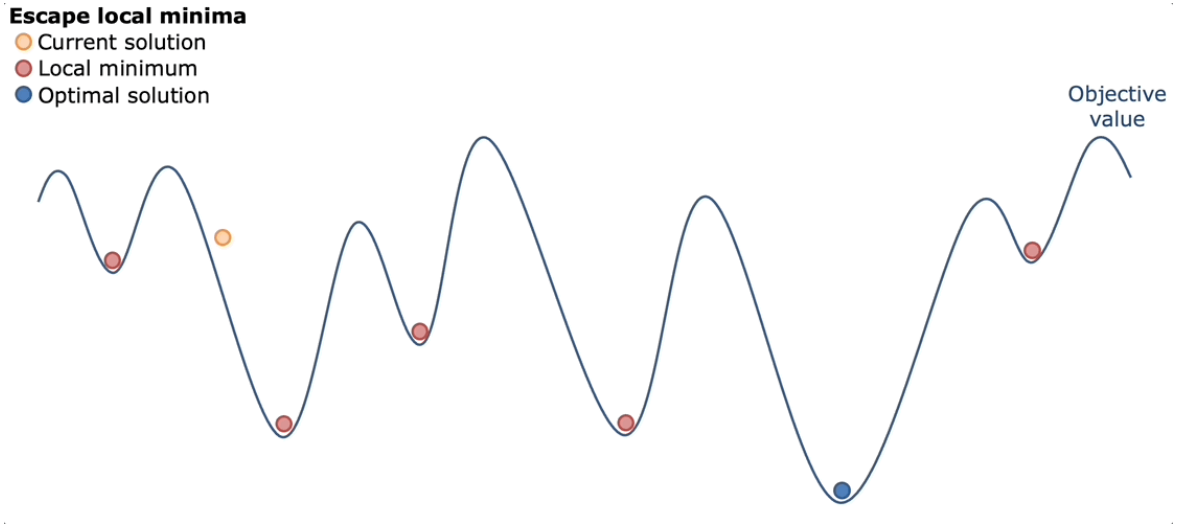
Late (cold): few bad moves accepted. Algorithm behaves like hill climbing, settles in.

Simulated annealing: the intuition

The metallurgy analogy (Kirkpatrick, Gelatt, Vecchi 1983)

Heat metal to high temperature, then cool slowly. At high temperatures, atoms vibrate energetically and can rearrange freely. As temperature drops, motion settles into a low-energy crystalline structure - a near-global minimum of the energy function.

Apply this to search: start by accepting many bad moves, gradually restrict to only good ones.



Simulated annealing: pseudocode (AIMA Fig. 4.8)

```
function SIMULATED-ANNEALING(problem, schedule):
    current  <-  problem.INITIAL_STATE

    for t = 1, 2, 3, ... :
        T  <-  schedule(t)
        if T = 0:  return current

        next  <-  a random neighbour of current

        delta_E  <-  VALUE(next) - VALUE(current)

        if delta_E > 0:
            current  <-  next
        else:
            current  <-  next  with probability
                                 e^(delta_E / T)
```

Cooling schedule
T starts high, decreases over time. The schedule() function defines the cooling rate.

T = 0 means stop
When the schedule cools to zero, behavior becomes deterministic - or we declare done.

Random neighbour
Unlike hill climbing's steepest ascent, SA picks one random neighbour to consider.

Compute change in value

delta_E > 0 means next is better. delta_E < 0 means next is worse.

Always accept better
Better neighbours are taken unconditionally, like hill climbing.

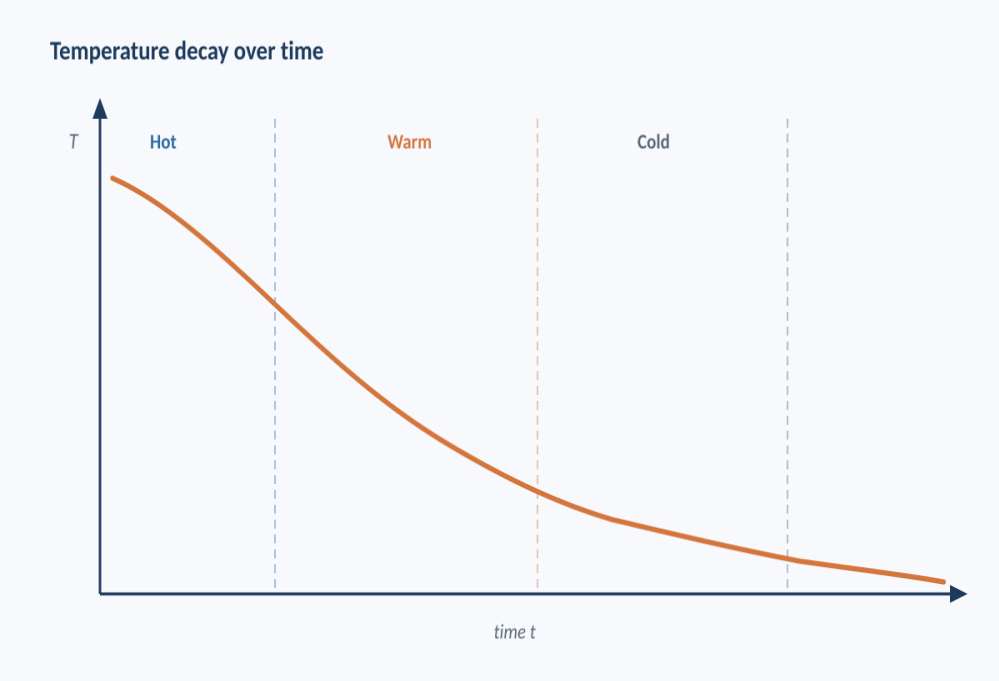
Probabilistically accept worse

Worse neighbours accepted with probability e^(delta_E/T). High T or small delta_E gives high probability. Low T or large delta_E gives low probability.

How temperature shapes the search

Beyond Jarrar's slides

The schedule controls when SA behaves like a random walk versus like hill climbing.



Behaviour by temperature regime

Hot (high T)

e^(ΔE/T) close to 1. Almost every move accepted, good or bad. Behaves like a random walk. Explores broadly.

Warm (moderate T)

Probability discriminates. Small bad moves accepted often; large bad moves rarely. The sweet spot for escaping local maxima while still making progress.

Cold (low T)

e^(ΔE/T) close to 0 for any bad move. Only better neighbours accepted. Behaves like hill climbing. Converges to a peak.

Simulated annealing: properties

Complete

Yes, with proper cooling schedule (theoretical).

Geman-Geman (1984): with T(t) >= c/log(t+2), SA finds the global optimum with probability 1 as t -> infinity. In practice, faster schedules sacrifice this guarantee for runtime.

Optimal

Yes in the limit of infinite time and slow cooling.

Same theoretical result. Practical SA returns a 'good' solution, not necessarily globally optimal. The trade-off: practical SA cools fast and may settle in a local optimum.

Time

Depends on the schedule. Often slow but tunable.

*Total iterations = T_max steps in the schedule * iterations per step. Often 10^4 - 10^6 iterations. Each iteration evaluates one random neighbour - cheap. Total still less than systematic search.*

Space

O(1) - one current state.

Like hill climbing, SA keeps only one state plus the temperature value. Constant memory regardless of problem size. The defining feature of local search.

03

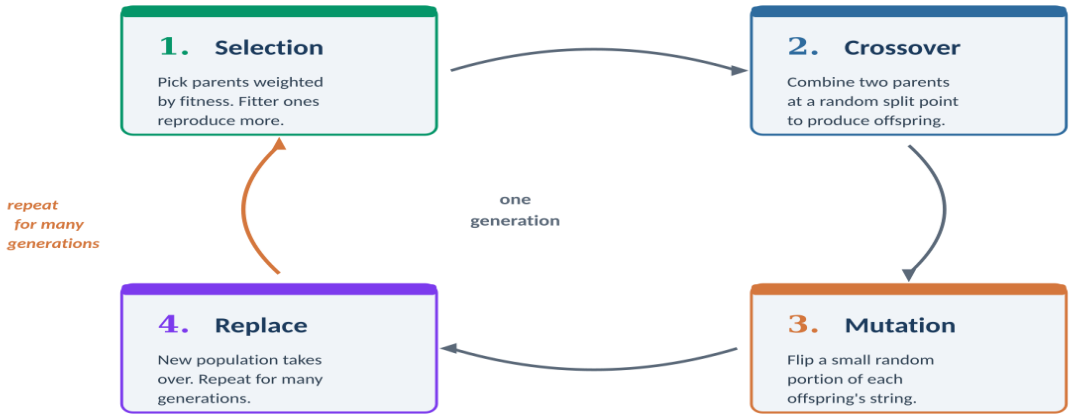
Genetic algorithms

A population evolves. Selection, crossover, mutation. Inspired by biology, used everywhere.

Genetic algorithms: the cycle

The big idea (Holland, 1975)
A population of candidate solutions evolves over generations. Better solutions survive and reproduce. Offspring combine traits of two parents. Random mutation introduces variation.

Like simulated annealing, but with a whole population instead of one state.



When to stop

Termination triggers:

- fitness threshold reached
- max generation count hit
- improvement stalls

Return the best individual seen.

Why one queen per column?

This is a formulation choice, not a game rule. The choice shrinks the search space.

The 8-queens rule

The only rule of 8-queens is: **no two queens attack each other**. Nothing says one queen per column. But any valid solution has at most one queen per column (otherwise two queens in the same column would attack). So we can **bake that constraint into the representation.**

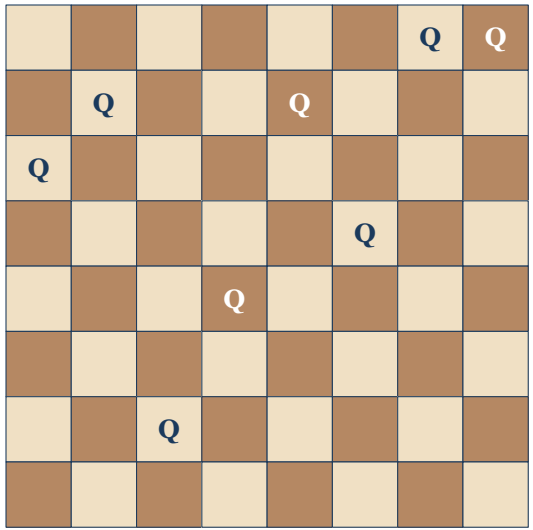
Formulation	Constraints	State space size	Conflicts still possible
8 queens, anywhere on 64 squares	None	$C(64,8) \approx 4.4$ billion	row, column, diagonal
One queen per column (chromosome)	Column	$8^8 \approx 16.7$ million	row, diagonal
Permutation (one per col, distinct rows)	Column, row	$8! = 40{,}320$	diagonal only

Local search picks the middle option. Small enough to search effectively. Large enough that crossover and mutation can still explore. *A common pattern: bake constraints into the representation when you can.*

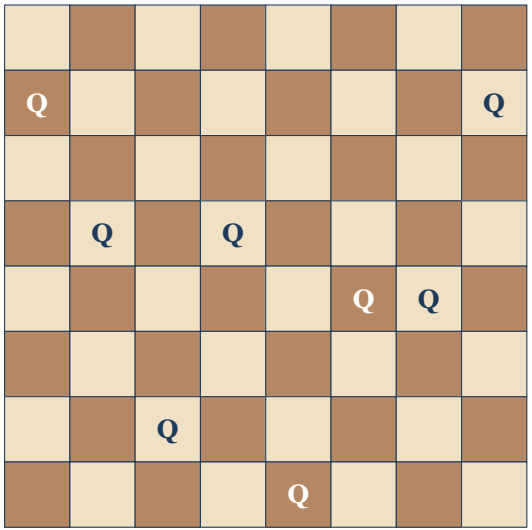
8-queens as chromosomes

Each chromosome is an 8-digit string. Position c gives the row of the queen in column c.

Chromosome 1: `32752411`



Chromosome 2: `24748552`



Reading a chromosome: the c-th digit gives the row of the queen in column c. *Crossover splits these strings at a random point and swaps tails. Mutation flips a single digit.*

Fitness and selection

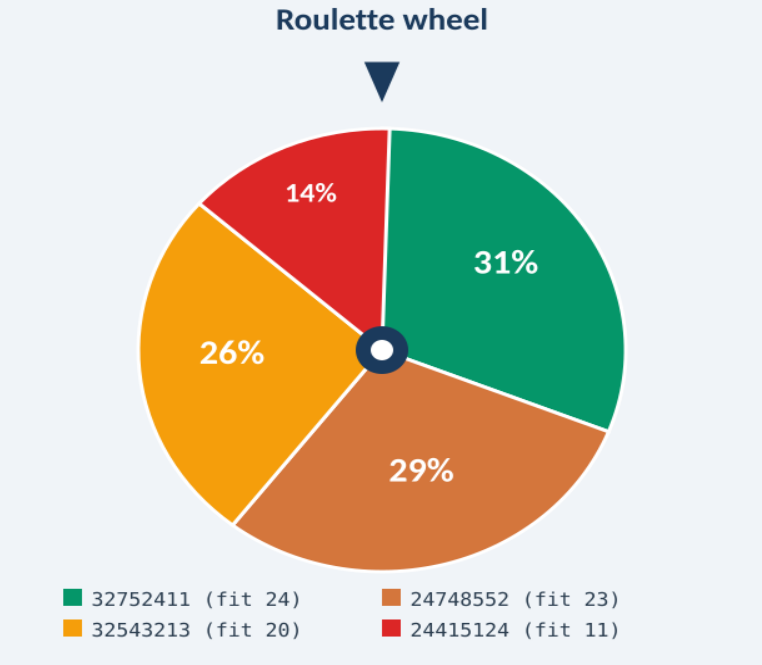
*Fitness = number of non-attacking pairs. Max = 8 * 7 / 2 = 28. Higher is better.*

Chromosome	Fitness	Selection probability
32752411	24	24 / 78 = 31%
24748552	23	23 / 78 = 29%
32543213	20	20 / 78 = 26%
24415124	11	11 / 78 = 14%

Why max fitness = 28

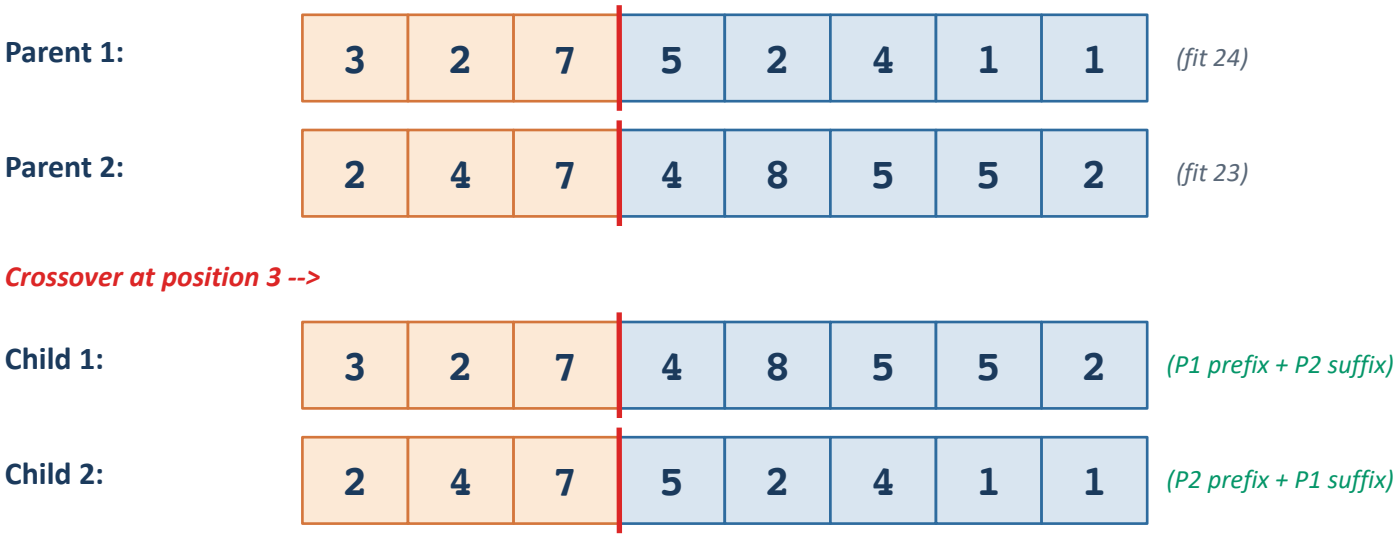
Total possible queen pairs = `8 choose 2 = 28`. Fitness counts pairs **not attacking**. A solution has all 28 safe.

Equivalently: fitness = 28 - h.



Crossover: combining two parents

Pick a random split point. Take prefix from parent 1, suffix from parent 2. Vice versa for the second child.



Mutation step

With small probability per digit (typically 0.01-0.1), flip a digit to a random value 1-8. *Example:* `32748552` becomes `32748152` (digit 6 flipped from 5 to 1).

GA pseudocode (AIMA Fig. 4.11)

```
function GENETIC-ALGORITHM(population,
                           fitness):

    repeat:
        weights  <-  map(fitness, population)
        pop2  <-  []

        for i = 1 to SIZE(population):
            p1, p2  <-  WEIGHTED-RANDOM-
                        CHOICES(population, weights, 2)
            child  <-  REPRODUCE(p1, p2)
            if (small random prob):
                child  <-  MUTATE(child)
            add child to pop2

        population  <-  pop2

    until fit enough or time up

    return best individual in population
```

Compute fitness for all

Evaluate every individual in the population. Used to weight selection.

Pick weighted parents

WEIGHTED-RANDOM-CHOICES samples 2 parents with probability proportional to fitness. Roulette wheel.

Crossover

Mutate sometimes

With small probability, perturb the child.

Build new generation

Repeat until done

Stop on time budget or when fitness reaches a threshold.

Genetic algorithms: properties

Complete

Probabilistically. With mutation and enough generations, eventually explores everywhere.

Mutation guarantees any state is reachable. But practical GA may run out of time long before discovering the right region. Like SA: theoretical limit, practical caveat.

Optimal

No. Returns the best found, not necessarily globally best.

GA is a heuristic. It often produces excellent solutions but rarely the proven optimum. Use when 'very good' is acceptable, not when 'provably best' is required.

Time

O(generations * population * fitness_evaluation).

Each generation: SIZE evaluations + SIZE selections + SIZE crossovers + occasional mutations. Total work scales linearly in generations and population. Fitness cost dominates for complex problems.

Space

O(population). Larger than HC/SA but still very small.

Maintains a population of ~50-1000 individuals. Each is a state of constant size. Much smaller than A's frontier on the same problems. The 'k' in beam search generalized to evolution.*

04

Comparison and wrap-up

When to use which algorithm.

Comparison of local search algorithms

Algorithm	Complete?	Optimal?	Memory	Escapes local max?
Hill-climbing	No	No	$O(1)$	No
HC with restart	Probabilistic	Probabilistic	$O(1)$	Yes (by restart)
Simulated annealing	Yes (slow cool)	Yes (slow cool)	$O(1)$	Yes (probabilistic)
Genetic algorithm	Probabilistic	No	$O(pop)$	Yes (mutation + diversity)

When to use which

HC: fast, simple, when the landscape is easy. **HC+restart:** moderate problems, when retries are cheap.
SA: rugged landscape, optimality matters. **GA:** large discrete spaces, multiple regions worth exploring, problem decomposes into sub-patterns.

Summary and references

Key takeaways

Local search tackles optimization problems where the goal is a state, not a path. Memory O(1) or O(population), feasible on enormous problems.

Hill climbing is the simplest. Greedy. Gets stuck. Fix with random restart.

Simulated annealing escapes local maxima by accepting worse moves probabilistically. Cooling schedule controls behaviour.

Genetic algorithms evolve a population via selection, crossover, mutation. Good for discrete structured problems.

References

Primary textbook

Russell & Norvig (2021). Artificial Intelligence: A Modern Approach. 4th Ed.
Section 4.1 (Local search), Figure 4.5 (HC), 4.8 (SA), 4.11 (GA).

Original papers

Kirkpatrick, Gelatt, Vecchi (1983). Optimization by Simulated Annealing. Science 220.

Holland, J. (1975). Adaptation in Natural and Artificial Systems. MIT Press.

Geman, S. & Geman, D. (1984). Stochastic Relaxation. IEEE PAMI 6.

Course materials

M. Jarrar (2020). Local Search Algorithms. Birzeit University.
jarrar.info/courses/AI/Jarrar.LectureNotes.Ch4.LocalSearch.pdf