Martin Ombura Jr.

Evolutionary Algorithms + NP-Hard Problems

Using Evolutionary Algorithms to Quickly Approximate NP-Hard Problems with Rust

RustLabIT 2025

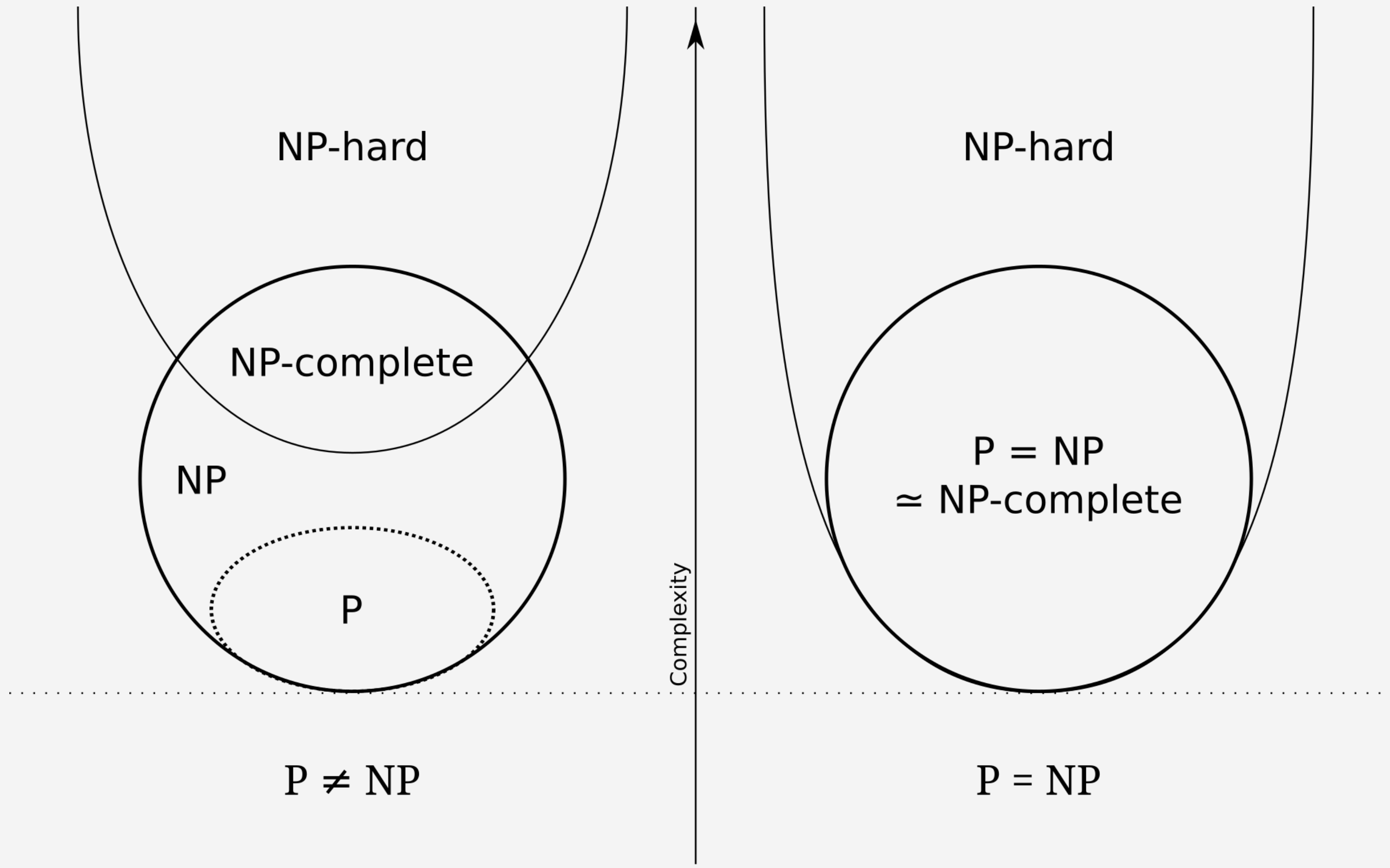
ToC

Computational Complexity Theory

P=NP???

1. P & NP
2. NP Hard
3. NP-Complete

Computational Complexity - Definitions



Computational Complexity - Definitions

P

- The class of decision problems solvable by a deterministic algorithm in polynomial time (O(n^k) for some constant k).

Think: Problems with algorithms we know run “efficiently.”

Examples:

- Shortest Path (Dijkstra)
- Minimum Spanning Tree, Max Flow, Bipartite
- Sorting, String Matching (KMP) etc

NP

- Decision problems for which a “yes” answer has a polynomial-size certificate verifiable in polynomial time. Equivalently, solvable in polynomial time by a nondeterministic machine.

- **Basically**: If someone hands you a solution, you can check it fast.

- **Examples** (classic):
  - Boolean satisfiability (SAT),
  - Traveling Salesman (decision): “Is there a tour ≤ K?”

[1]https://en.wikipedia.org/wiki/NP-completeness

NP-Complete

"NP-complete" is short for **"nondeterministic polynomial-time complete"**

- NP-complete problems are the hardest of the problems to which solutions can be verified quickly (in polynomial time).
- Finding the solutions however takes longer than polynomial (often exponential)
- **Examples**
  - Boolean SAT
  - Hamiltonian Cycles,
  - TSP (decision format)
  - 3-Coloring Problem
  - Graph Coloring

Somewhat more precisely, a problem is NP-complete when:

1. It is a decision problem,
   a. for any input the output is either **"yes"** or **"no"**.
   b. If **"yes"**, this can be demonstrated through the existence of a short (polynomial length) solution.
2. The correctness of each solution can be verified quickly (namely, in polynomial time)
3. If any NP-complete problem is in P, then P = NP.

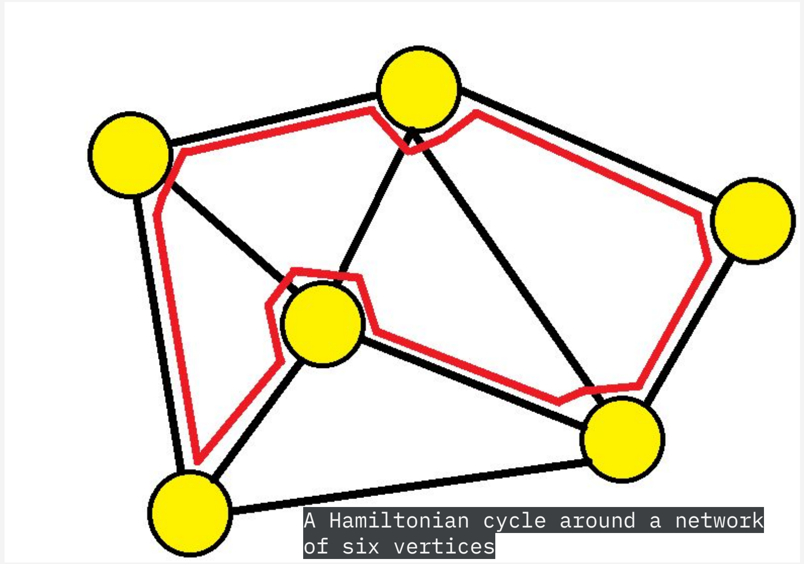
https://en.wikipedia.org/wiki/NP-completeness

NP-Complete Examples: Hamiltonian Cycle Problem

Checks if a directed/undirected graph G, contains a “Hamiltonian Path”. I.e a Path that visits every vertex in the graph exactly once.

- A Hamiltonian path that starts and ends at adjacent vertices can be completed by adding one more edge to form a Hamiltonian cycle, and removing any edge from a Hamiltonian cycle produces a Hamiltonian path

- The computational problems of determining whether such paths and cycles exist in graphs are **NP-complete**


A Hamiltonian cycle around a network of six vertices

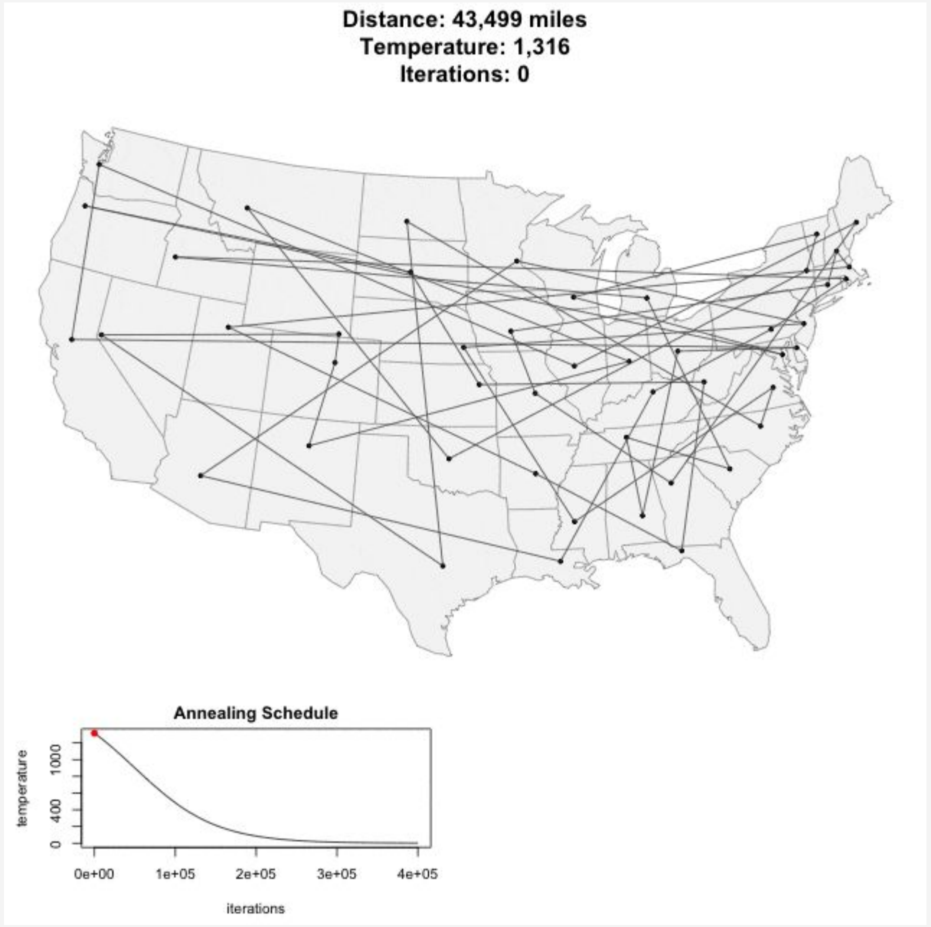
https://en.wikipedia.org/wiki/Hamiltonian_path
https://www3.cs.stonybrook.edu/~skiena/combinatorica/trias/index.html

NP-Complete Examples: Traveling Salesman Decision

- The travelling Salesman problem (TSP) asks the following question:

"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

NP-Complete:
“Given distances and K, is there a tour ≤ K?”

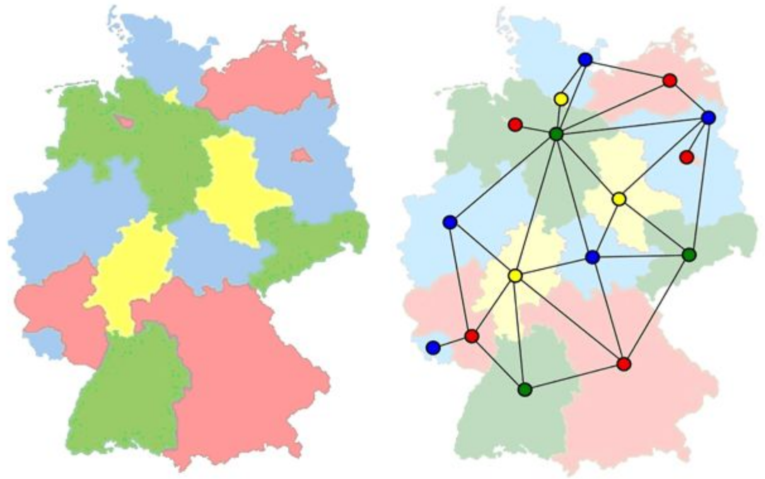


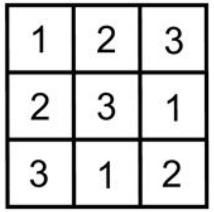
https://en.wikipedia.org/wiki/Travelling_salesman_problem
https://toddwschneider.com/posts/traveling-salesman-with-simulated-annealing-r-and-shiny/

NP-Complete Examples: Graph Coloring

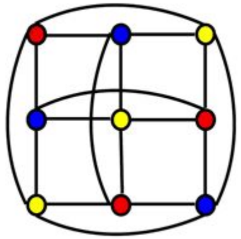
- A graph theory problem that attempts to assign colors (or labels) to vertices of a graph under a given set of constraints
    - E.g. no two adjacent vertices are of the same color

- Has seen applications in other domains e.g. Sudoku





1	2	3
2	3	1
3	1	2



A Hamiltonian cycle around a network of six vertices

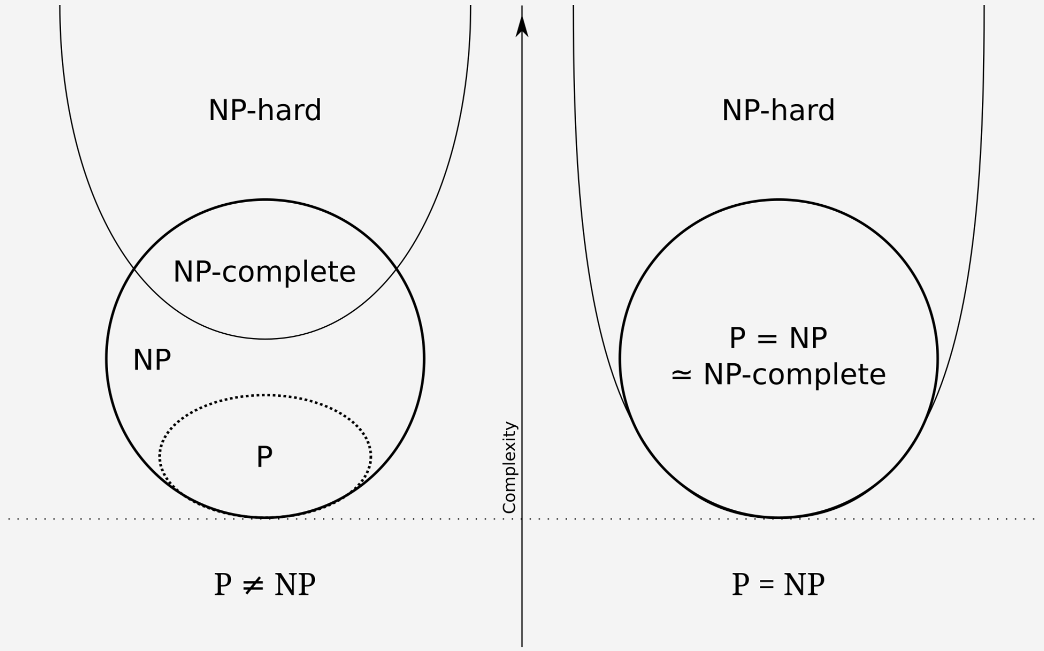
https://en.wikipedia.org/wiki/Graph_coloring
https://phys.org/news/2009-02-kind-scientists-algorithm-previously-unsolvable.html

NP-Hard

- Class of problems which are at least as hard as the hardest problems in NP. Problems that are NP-hard do not have to be elements of NP; indeed, they may not even be decidable.

- NP-hard problems **need not be** in NP (they might not even be decision problems, or could be undecidable

- **Examples**
    - Traveling Salesman Optimization Problem
    - Knapsack Optimization Problem
    - Graph Coloring (Register Allocation in Compilers)



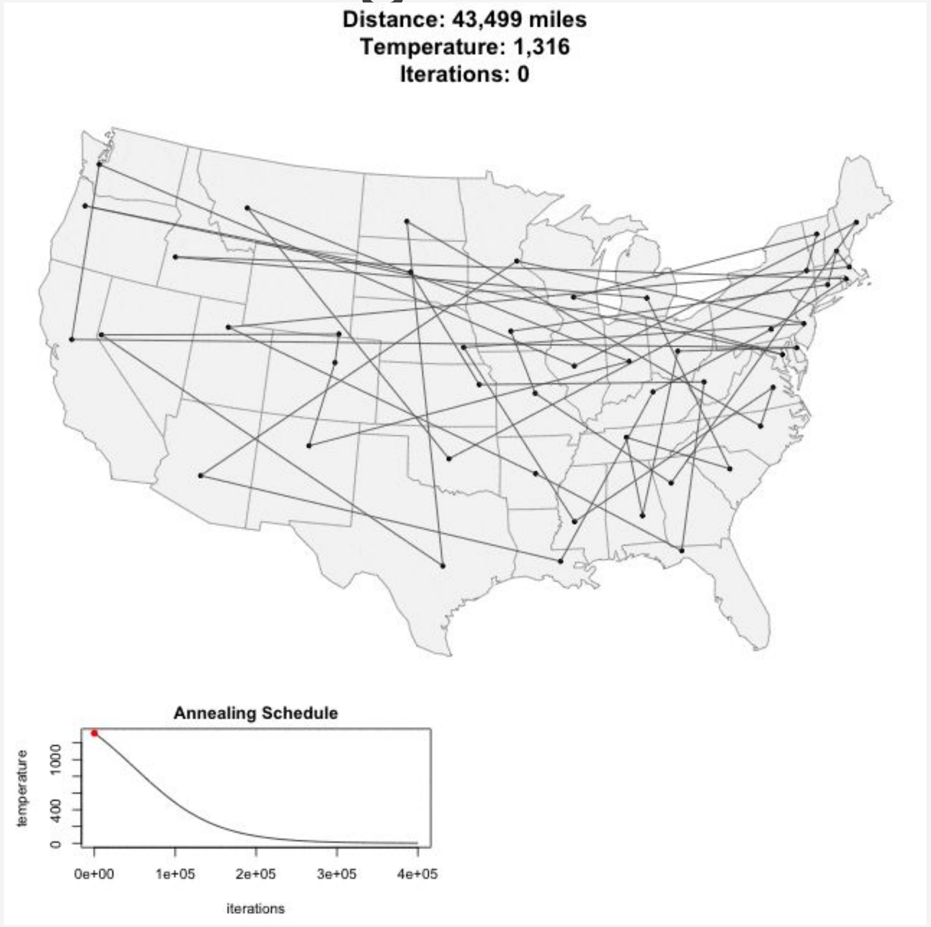
NP-Hard Examples: Traveling Salesman

- **The travelling salesman problem (TSP) asks the following question:**

"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

- **Variants**
  - The travelling purchaser problem
  - The vehicle routing problem
  - The ring star problem

are three generalizations of TSP.

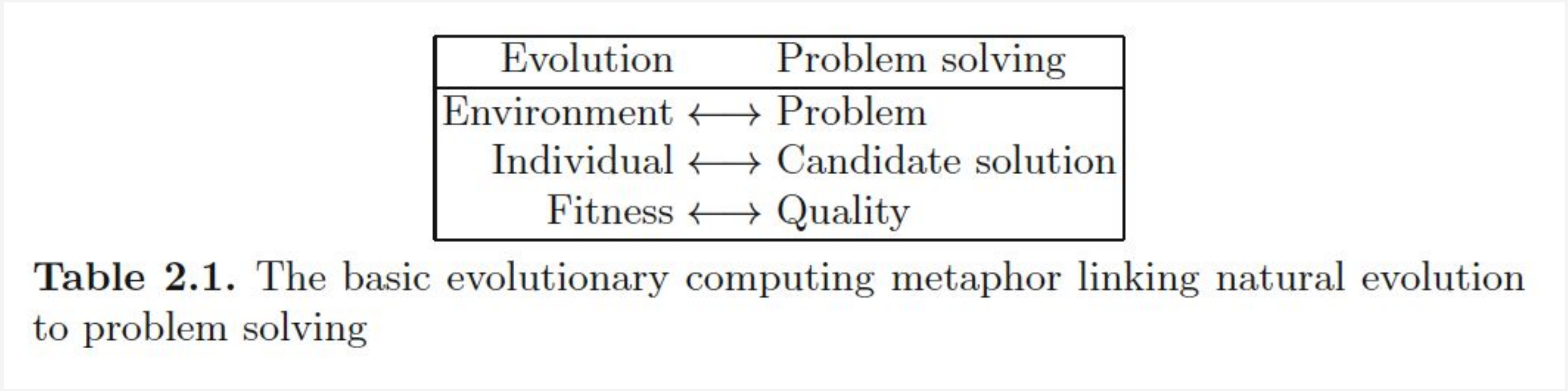


https://en.wikipedia.org/wiki/Travelling_salesman_problem
https://phys.org/news/2009-02-kind-scientists-algorithm-previously-unsolvable.html

1. Evolutionary Algorithms
2. Genetic Algorithms
3. Evolutionary Process

Evolutionary Algorithms

Evolutionary Algorithms



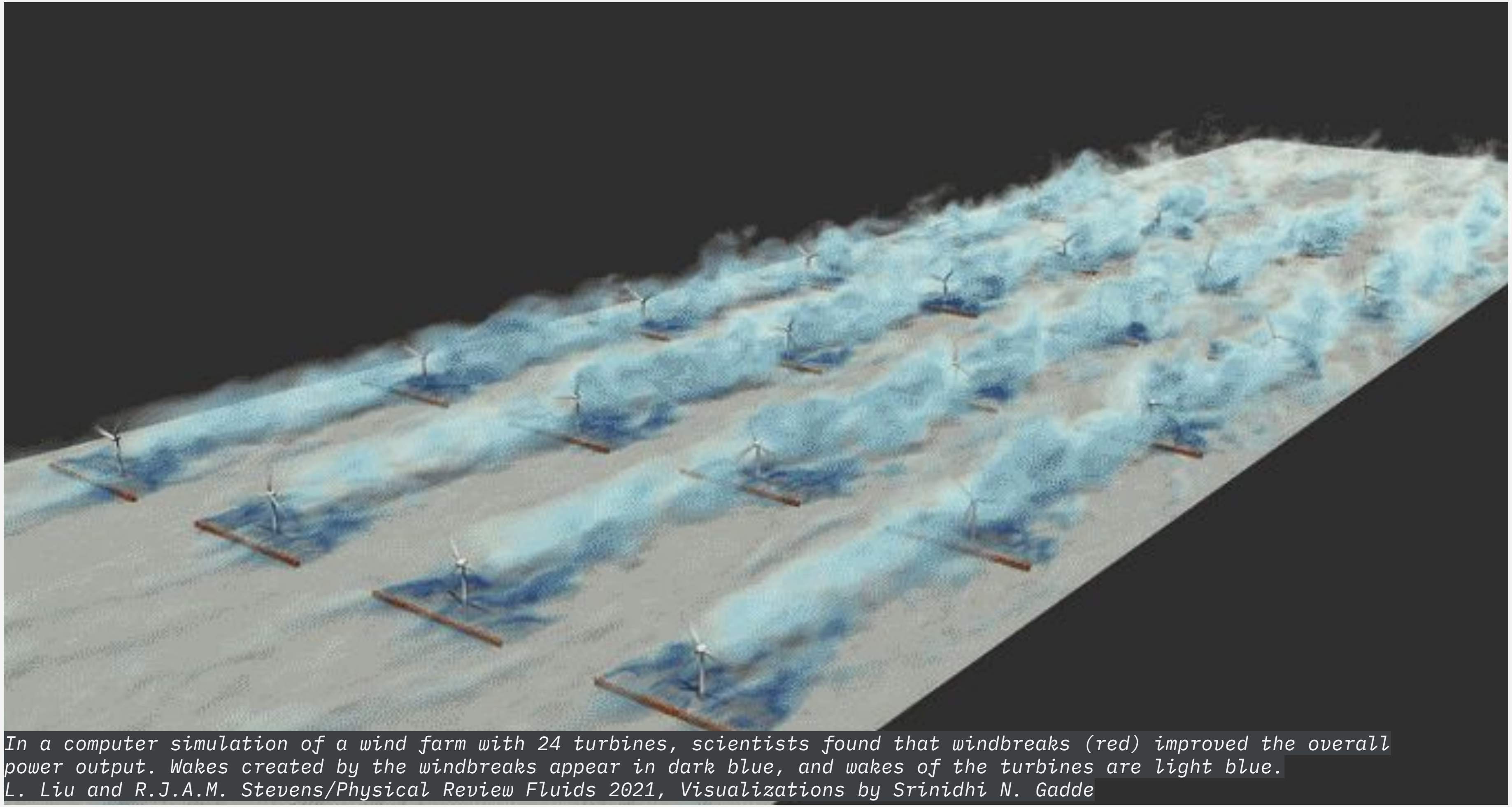
Evolution		Problem solving
Environment	$\longleftrightarrow$	Problem
Individual	$\longleftrightarrow$	Candidate solution
Fitness	$\longleftrightarrow$	Quality

Table 2.1. The basic evolutionary computing metaphor linking natural evolution to problem solving

- Evolutionary algorithms (EAs), are a family stochastic search methods, inspired by natural biological evolution, that operate on a population of potential solutions using the principle of survival of the fittest to produce better and better approximations to a solution [1]

[1]Automated Antenna Design with Evolutionary Algorithms - http://alglobus.net/NASAwork/papers/Space2006Antenna.pdf

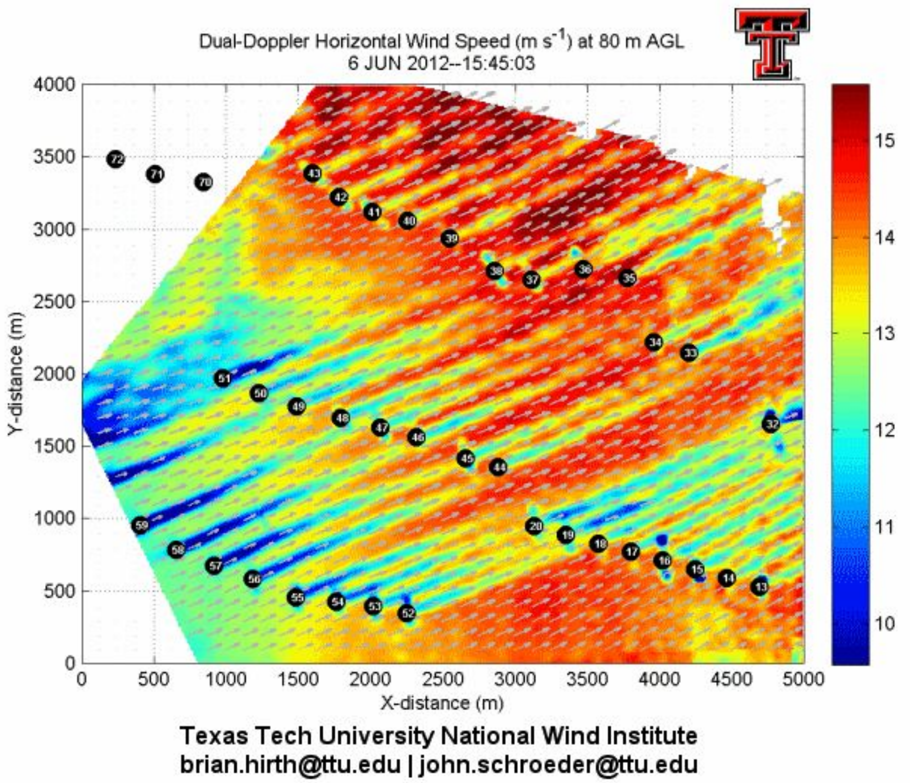
Wind Turbine - Wake Simulations

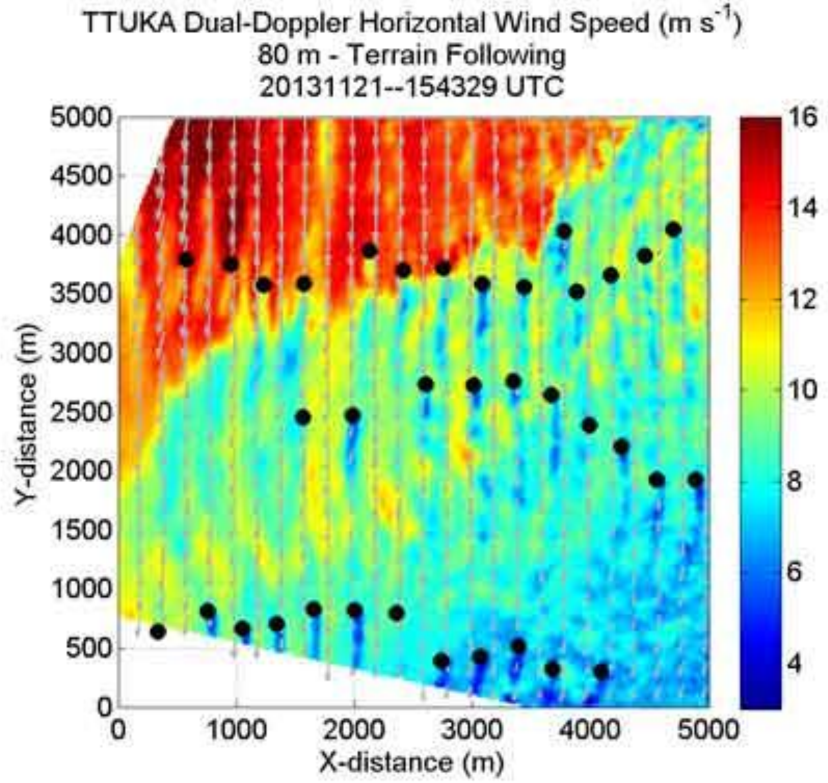


In a computer simulation of a wind farm with 24 turbines, scientists found that windbreaks (red) improved the overall power output. Wakes created by the windbreaks appear in dark blue, and wakes of the turbines are light blue.
L. Liu and R.J.A.M. Stevens/Physical Review Fluids 2021, Visualizations by Srinidhi N. Gadde

[1]https://www.sciencenews.org/article/wind-farm-windbreak-turbine-speed-power-output
[2] Radar Wake Research - h

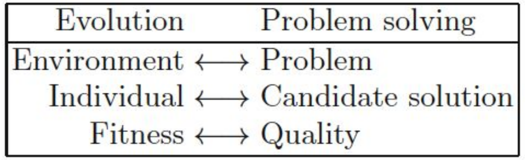
Wind Turbine - Wake Simulations





[1]Automated Antenna Design with Evolutionary Algorithms - https://wes.copernicus.org/articles/9/2113/2024
[2] Radar Wake Research - https://www.depts.ttu.edu/nwi/research/radar-research/index.php

Evolutionary Landscape



Evolution		Problem solving
Environment	⟷	Problem
Individual	⟷	Candidate solution
Fitness	⟷	Quality

Table 2.1. The basic evolutionary computing metaphor linking natural evolution to problem solving

Environment

-

Individual (Solution)

- The notion of representation is critical in outlining how an individual will be modelled in code (GAs use bitstrings, GPs use DAGs)

Fitness Function

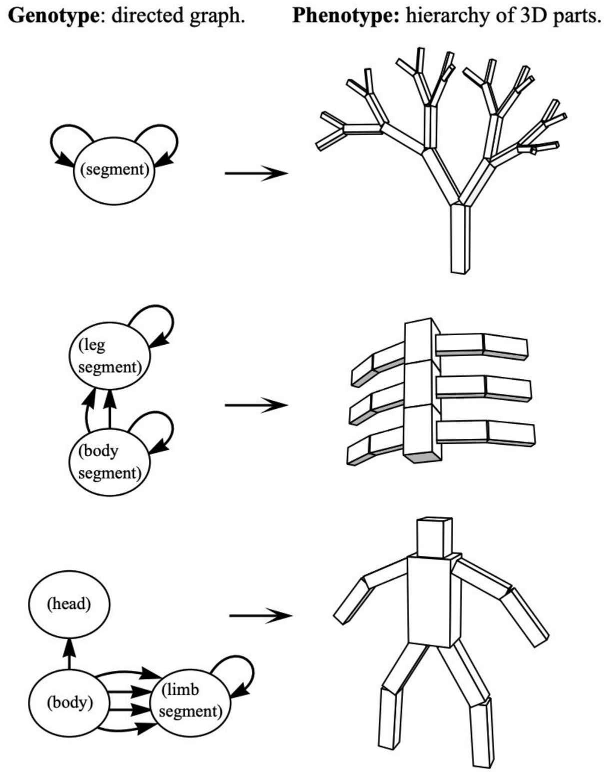

Genotype: directed graph.
Phenotype: hierarchy of 3D parts.
(segment)
(leg segment)
(body segment)
(head)
(body)
(limb segment)

Figure 1: Designed examples of genotype graphs and corresponding creature morphologies.

https://www.karlsims.com/papers/siggraph94.pdf
(Eiben & Smith, 2015) - Introduction to Evolutionary Computing

Genetic Algorithms

- **Definition**: Genetic Algorithms (GAs) are population-based, randomized search/optimization methods that iteratively evolve candidate solutions via selection, recombination, and mutation, guided by a fitness function.

- **Purpose**:
  - Produce good approximations quickly on tough combinatorial/continuous landscapes, explore large, deceptive search spaces, and give you great visuals of search dynamics.
  - GAs excel where objectives are
    - Black-box (simulation or test-stand), non-convex, multi-objective, and/or mixed-integer—and where broad design-space exploration is valuable.

- GA's act as heuristics. They do not prove anything about P vs NP and they do not "solve" NP-hard problems in the sense of guaranteed optimal, polynomial-time algorithms.

(Mitchell, 1996) - An introduction to Genetic Algorithms

NASA - JPL: ST5 Mission Antennae

- NASA needed a tiny X-band antenna with wide beamwidth, circular polarization, and adequate bandwidth, robust to arbitrary spacecraft orientation.

- Hand design was time-intensive and struggled with conflicting RF specs; a GA could search unconventional 3D shapes directly against a physics simulator (“black-box” fitness)

- When requirements changed late, the team re-evolved a compliant antenna in < 1 month by updating the fitness

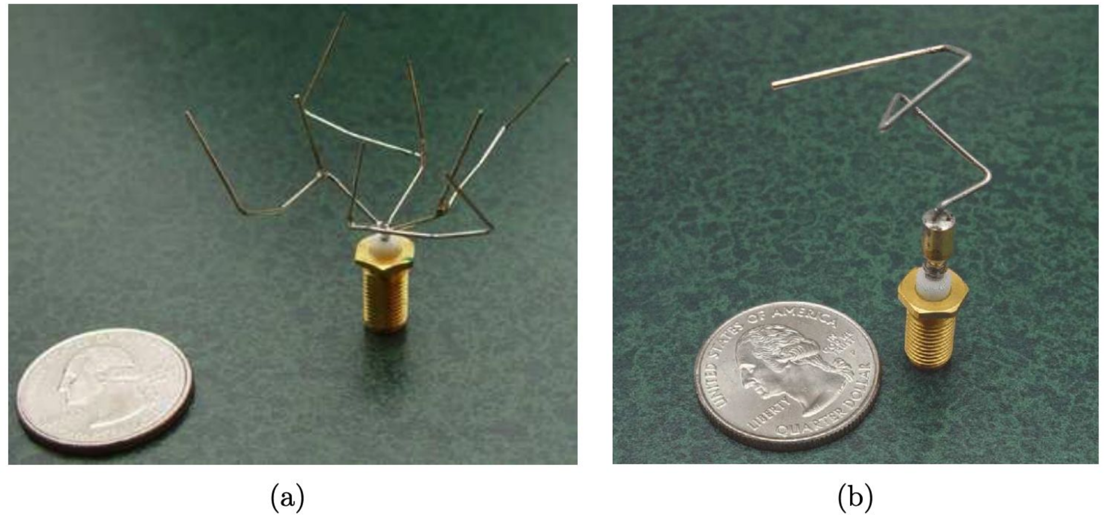


Figure 2. Photographs of prototype evolved antennas: (a) the best evolved antenna for the initial gain pattern requirement, ST5-3-10; (b) the best evolved antenna for the revised specifications, ST5-33-142-7.

[1]Automated Antenna Design with Evolutionary Algorithms - http://alglobus.net/NASAwork/papers/Space2006Antenna.pdf
https://ntrs.nasa.gov/citations/20060024675

Genetic Algorithms & NP-Hard

GA’s act as heuristics. **They do not prove anything about P vs NP** and they do not “solve” NP-hard problems in the sense of guaranteed optimal, polynomial-time algorithms. They provide robust means to quickly approximate solutions in even more varied and complex landscapes

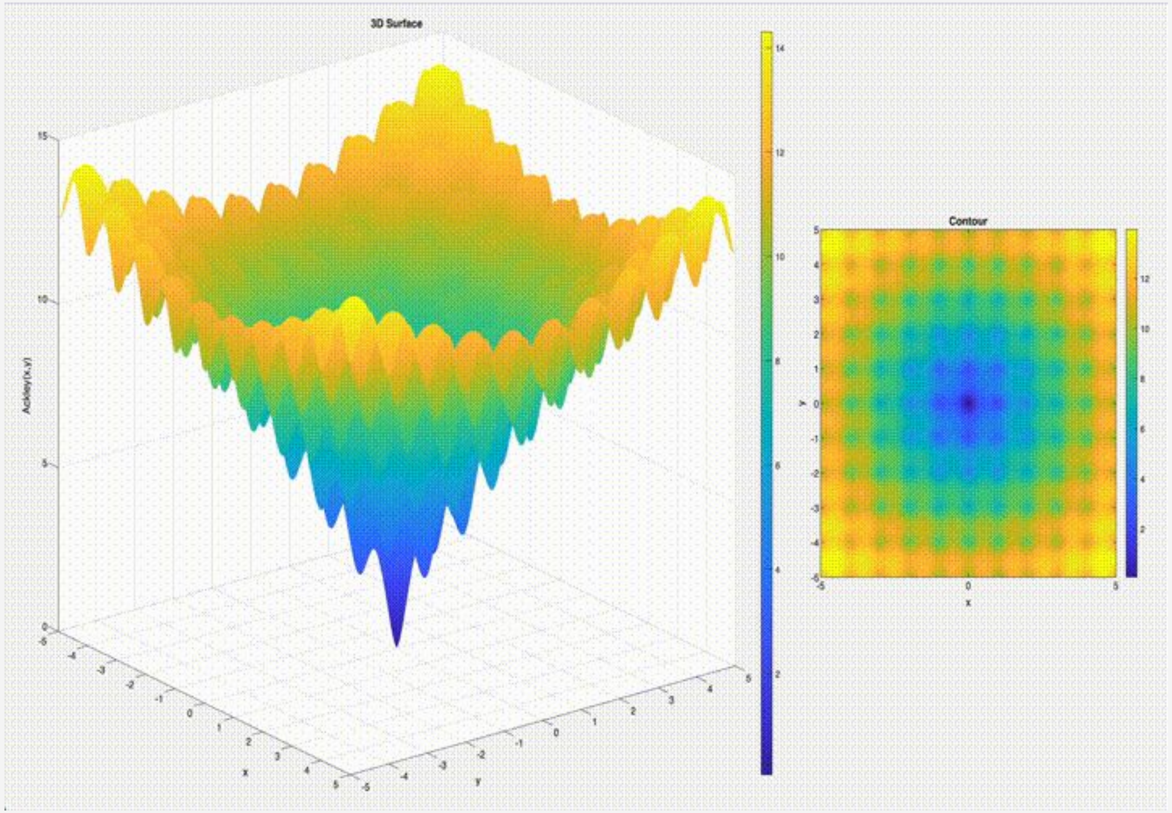
Optimization Functions

Ackley Function
Class: Multiple-Local-Minima

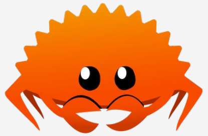
- Appears flat and unimodal from afar, yet hides many shallow traps close to the center.

GA Relevance

- Excellent benchmark for premature convergence—tests whether the population can escape local ripples
- Large mutation radii or self-adaptive σ strategies help traverse the plateau.
- Diversity preservation (niching, fitness sharing) prevents crowding in false minima.
- Hybrid local search (memetic GA) near the center accelerates fine tuning once diversity drops.
- Demonstrates how multimodality + deceptive smoothness challenge both exploitation



RUST! Where does it fit in?



- Throughput + safety. Zero-cost abstractions, no GC, predictable perf; ideal for many small allocations and tight loops.

- Determinism at scale. `rand_chacha` with explicit seeds per thread → reproducible runs

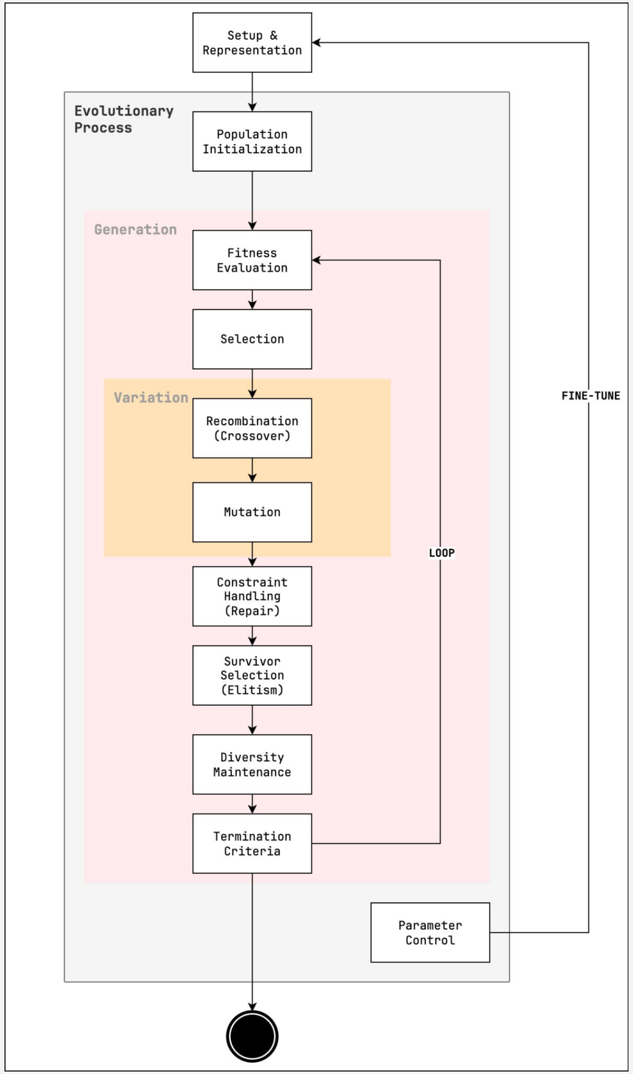
- Effortless parallelism. Fitness evaluation via `rayon::par_iter();` per-thread RNG; reduction for best-of-gen.

1. Population Initialization
2. Fitness Evaluation
3. Check
4. Parent Selection
5. Reproduction
6. Mutation
7. Natural Selection
8. Repeat

Evolutionary Process

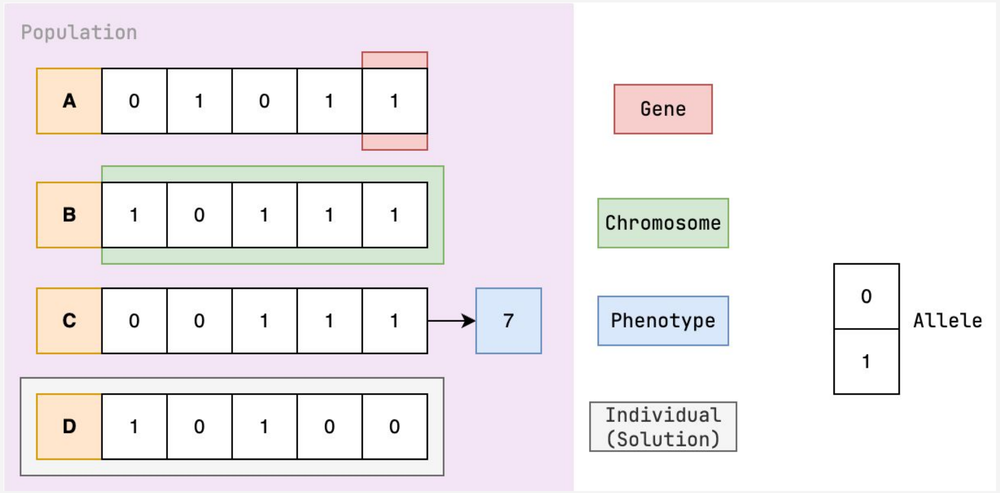
Evolutionary Process

1. Problem Setup & Representation
2. Population Initialization
3. Fitness Evaluation (per individual)
4. Selection (Choose parents)
5. Recombination (Produce offspring)
    a. Crossover (with probability p_c)
6. Mutation
    a. (applied per offspring, often with per-gene probability p_m)
7. Constraint Handling & Replacement
    a. Form the next generation)
8. Diversity Maintenance
    a. Avoid premature convergence)
9. Termination Criteria
10. Post-Run: What to return and how to finish well



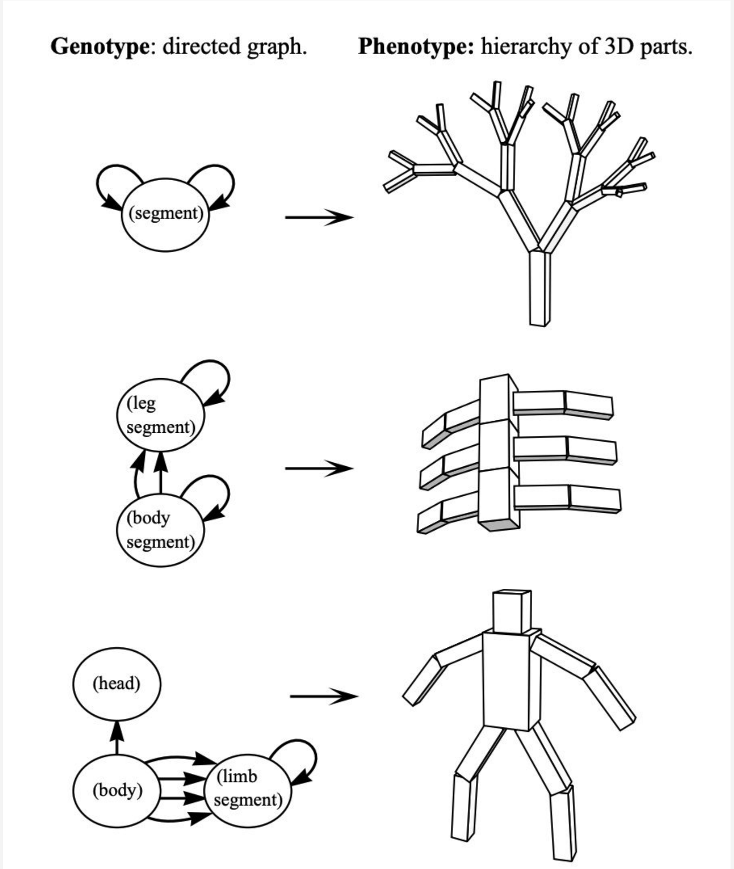
Quick Definitions

- **Gene:** The atomic unit of representation; a fixed position (locus) on a chromosome that holds part of the encoding of a solution.
- **Allele:** The value a gene assumes at its locus within a chromosome.
- **Chromosome:** An encoded candidate solution, typically a fixed-length string/array of genes (binary, integer, real, or symbolic).
    - Operated on by selection, crossover, and mutation; fitness evaluation is applied to the decoded form..
- **Phenotype:** The expressed solution in the problem domain obtained by decoding the genotype (e.g., the actual tour, schedule, or parameter vector)..



0. Problem Setup & Representation

- **Goal:** Optimize an objective under constraints (e.g., TSP distance; Ackley/Rastrigin functions).
- **Representation:**
  - A good encoding + decoding preserves meaningful building blocks and makes useful variation likely.
  - Representation dictates legal operators (PMX/OX for TSP; BLX-α/SBX for floats).
    - **TSP**: Vec<usize> of city IDs
    - **Ackley**: Vec<f32> of continuous ranges



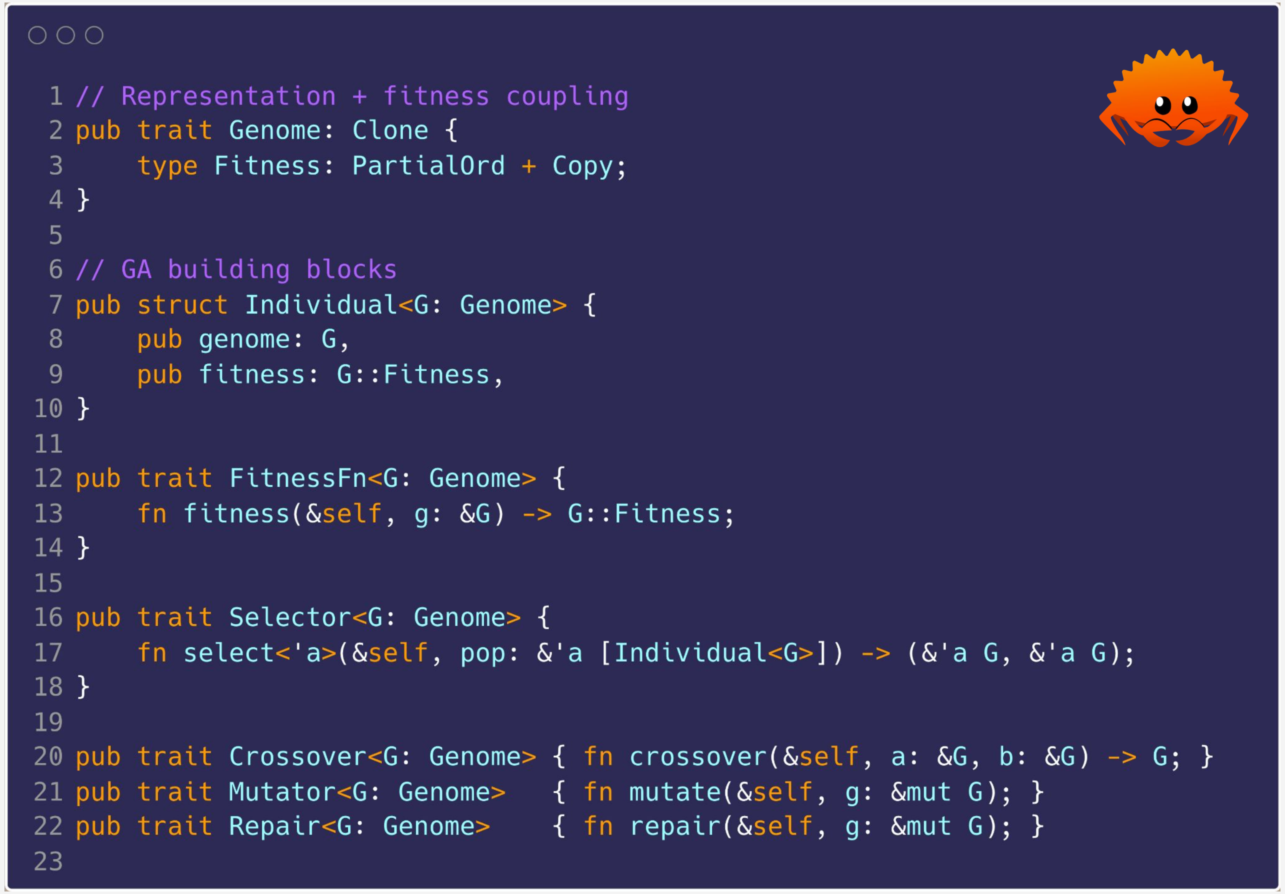
Genotype: directed graph.
Phenotype: hierarchy of 3D parts.
(segment)
(leg segment)
(body segment)
(head)
(body)
(limb segment)

Figure 1: Designed examples of genotype graphs and corresponding creature morphologies.

https://www.karlsims.com/papers/siggraph94.pdf

0. Problem Setup & Representation - Rust

- **Single engine, many domains:** Genome, FitnessFn, Selector, Crossover, Mutator, etc let us swap TSP (permutation) ⇔ continuous (Ackley/Rastrigin) without touching the loop.
- Rust monomorphizes generics, operator calls can also inline in the hot path



```rust
// Representation + fitness coupling
pub trait Genome: Clone {
    type Fitness: PartialOrd + Copy;
}

// GA building blocks
pub struct Individual<G: Genome> {
    pub genome: G,
    pub fitness: G::Fitness,
}

pub trait FitnessFn<G: Genome> {
    fn fitness(&self, g: &G) -> G::Fitness;
}

pub trait Selector<G: Genome> {
    fn select<'a>(&self, pop: &'a [Individual<G>]) -> (&'a G, &'a G);
}

pub trait Crossover<G: Genome> { fn crossover(&self, a: &G, b: &G) -> G; }
pub trait Mutator<G: Genome>   { fn mutate(&self, g: &mut G); }
pub trait Repair<G: Genome>    { fn repair(&self, g: &mut G); }

```

https://www.karlsims.com/papers/siggraph94.pdf

0. Problem Setup & Representation - TSP - Rust

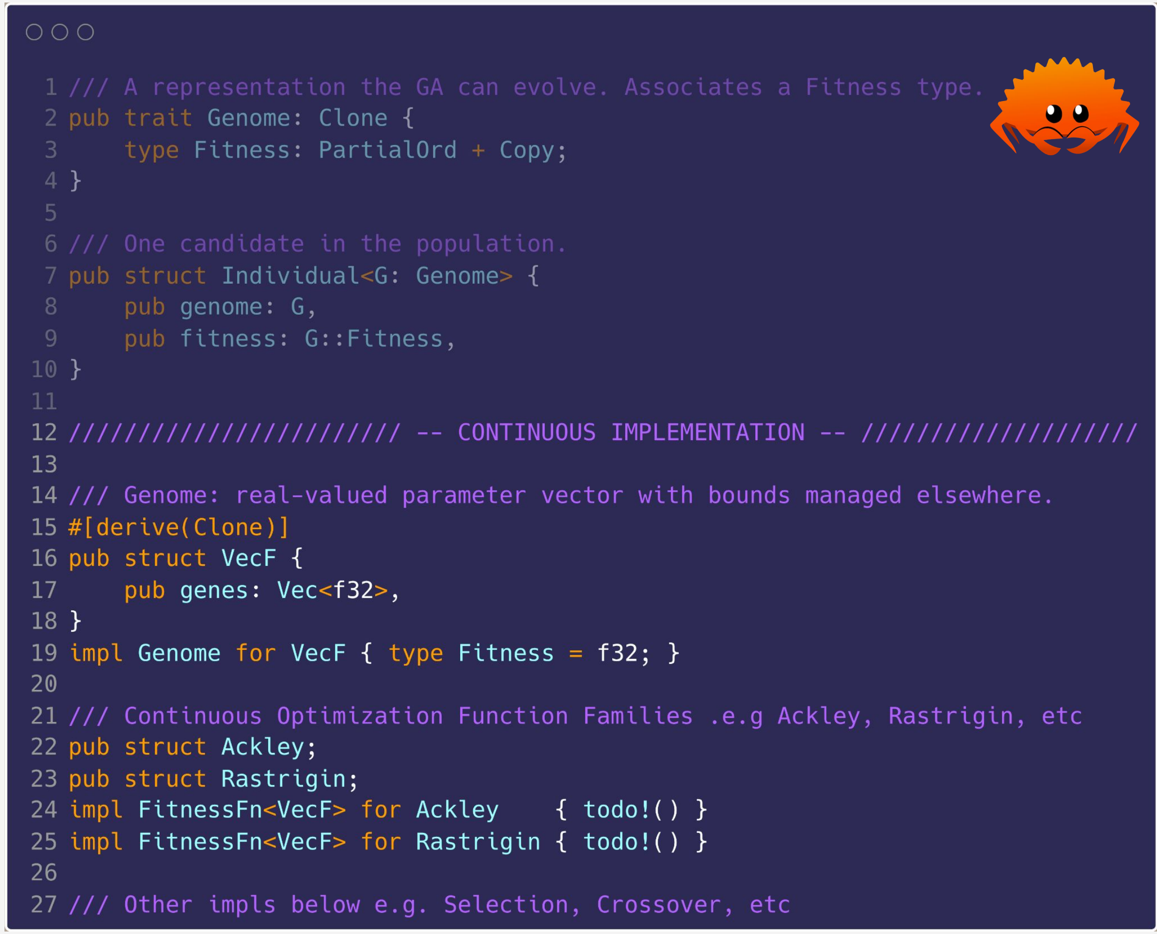
- **TSP:** TSP’s specific genome which we call a Tour can now monomorphically be encapsulated as a Vec<usize>

```rust
//rust
/// A representation the GA can evolve. Associates a Fitness type.
pub trait Genome: Clone {
    type Fitness: PartialOrd + Copy;
}

/// One candidate in the population.
pub struct Individual<G: Genome> {
    pub genome: G,
    // lower/higher is better (minimization/maximization) depends on problem
    pub fitness: G::Fitness,
}

//////////////////////// -- TSP IMPLEMENTATION -- ////////////////////////

/// Genome: a tour (permutation of city indices).
#[derive(Clone)]
pub struct Tour {
    pub order: Vec<usize>,
}
impl Genome for Tour { type Fitness = f64; }

/// Fitness descriptor: holds read-only data needed to score a tour.
pub struct TspFitness {
    pub distance_matrix: Vec<f32>, // flat row-major, len = n*n
    pub city_count: usize,
}
impl FitnessFn<Tour> for TspFitness { todo!() }

/// Other impls below...
```

https://www.karlsims.com/papers/siggraph94.pdf

0. Problem Setup & Representation - Continuous - Rust

- **Continuous Functions:** For our continuous family of optimization functions e.g. Ackley/Rastrigin,the genome which we call a VecF can now monomorphically be encapsulated as a Vec<f32>

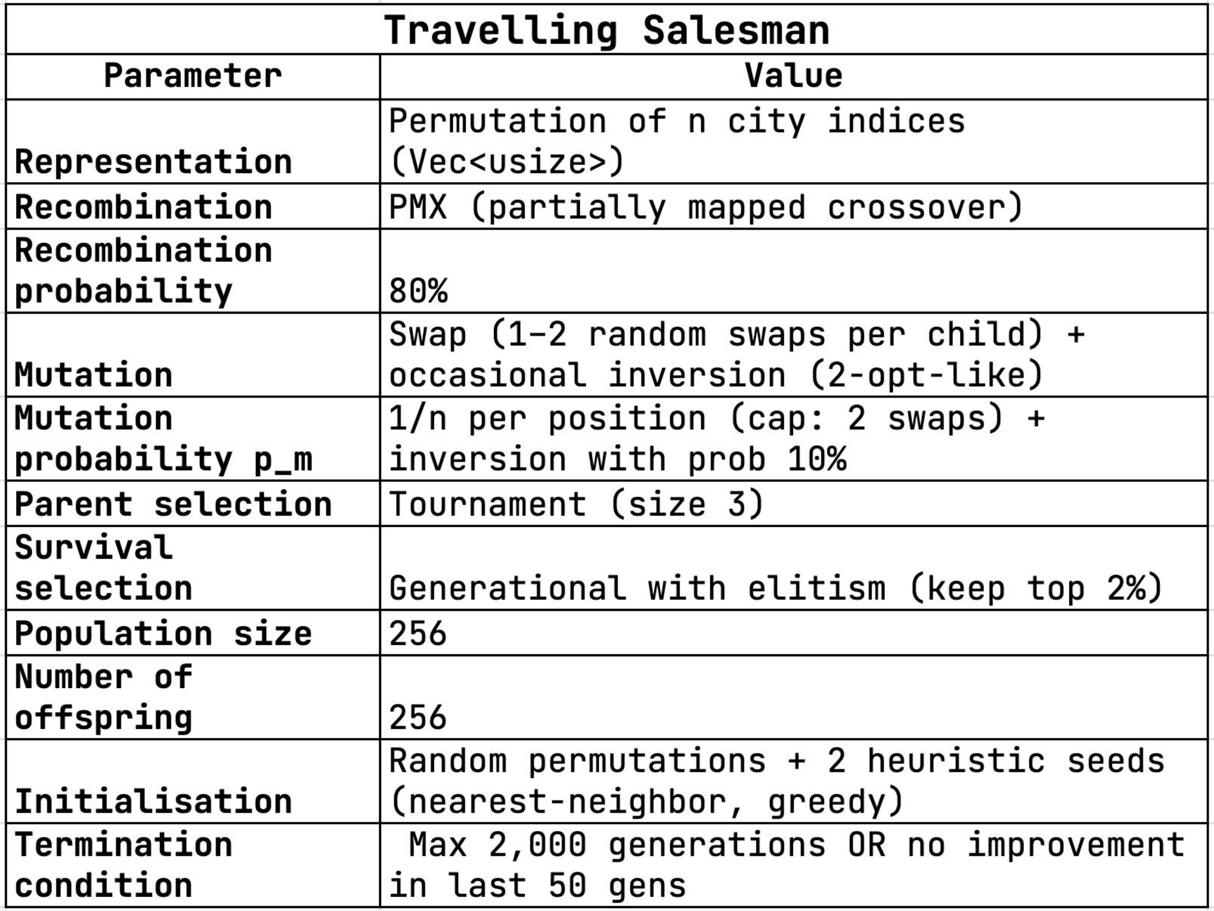


```rust
/// A representation the GA can evolve. Associates a Fitness type.
pub trait Genome: Clone {
    type Fitness: PartialOrd + Copy;
}

/// One candidate in the population.
pub struct Individual<G: Genome> {
    pub genome: G,
    pub fitness: G::Fitness,
}

/////////////////////// -- CONTINUOUS IMPLEMENTATION -- ///////////////////

/// Genome: real-valued parameter vector with bounds managed elsewhere.
#[derive(Clone)]
pub struct VecF {
    pub genes: Vec<f32>,
}
impl Genome for VecF { type Fitness = f32; }

/// Continuous Optimization Function Families .e.g Ackley, Rastrigin, etc
pub struct Ackley;
pub struct Rastrigin;
impl FitnessFn<VecF> for Ackley    { todo!() }
impl FitnessFn<VecF> for Rastrigin { todo!() }

/// Other impls below e.g. Selection, Crossover, etc
```

https://www.karlsims.com/papers/siggraph94.pdf

0. Problem Setup & Representation: Parameter Selection

- **Parameter Setup:** All the various parameters that influence how the evolutionary process unfolds



Travelling Salesman	
Parameter	**Value**
Representation	Permutation of n city indices (Vec<usize>)
Recombination	PMX (partially mapped crossover)
Recombination probability	80%
Mutation	Swap (1–2 random swaps per child) + occasional inversion (2-opt-like)
Mutation probability p_m	1/n per position (cap: 2 swaps) + inversion with prob 10%
Parent selection	Tournament (size 3)
Survival selection	Generational with elitism (keep top 2%)
Population size	256
Number of offspring	256
Initialisation	Random permutations + 2 heuristic seeds (nearest-neighbor, greedy)
Termination condition	Max 2,000 generations OR no improvement in last 50 gens

1. Population Initialization

- **Definition**: Create the initial population of candidate solutions.

- **Goal**: Early diversity prevents premature convergence; adding a few heuristic seeds can shorten time-to-good-solutions.

- **Examples:**
  - **TSP:** Random city permutations
    - One can use some exploitative greedy heuristics e.g. nearest-neighbor to seed quality.
  - **Continuous:** Uniform random within bounds.

- **Population size:** Tune for diversity vs. runtime. This can range from small (64) to large (2000+) depending on constraints

1. Population Initialization - Rust

Deterministic parallel RNGs:

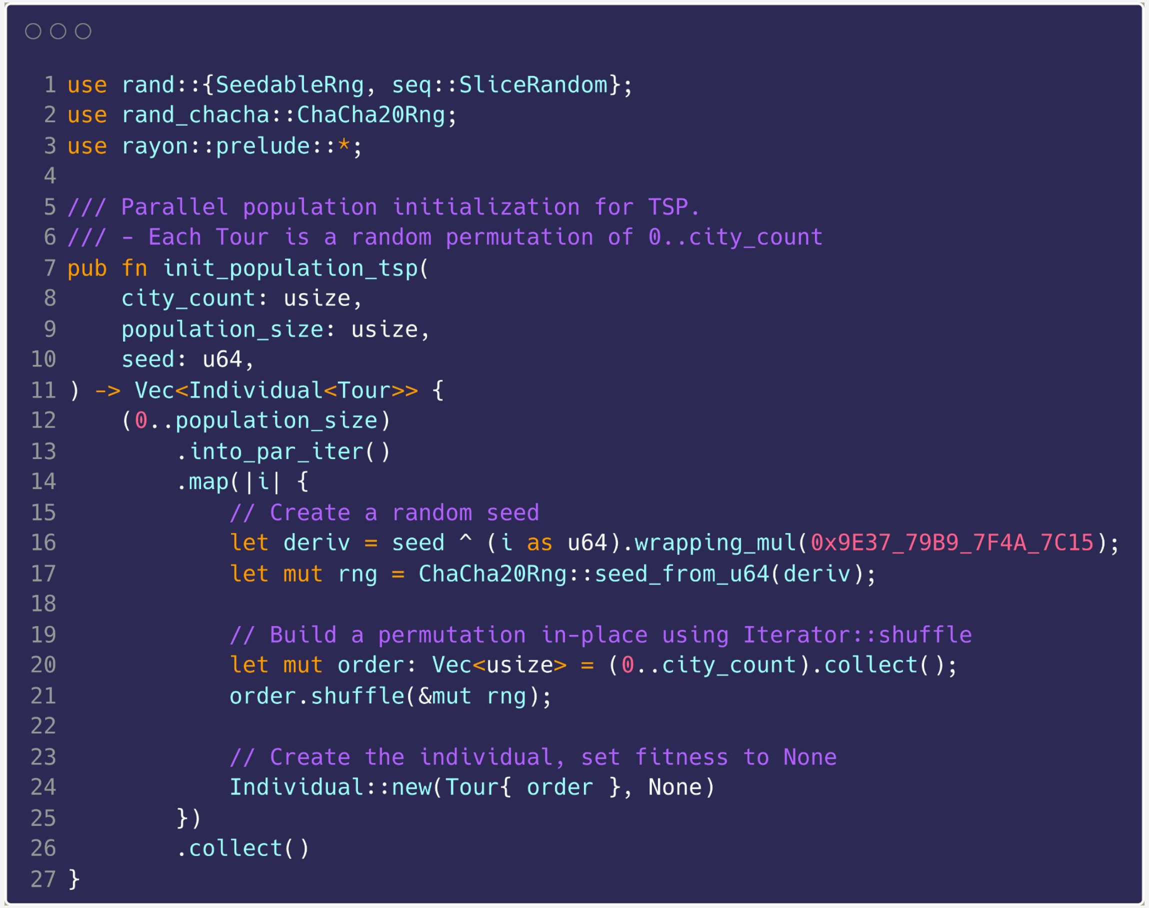
- Seed a fresh RNG per worker/individual (rand_chacha, rand_pcg, or rand_xoshiro) for thread-safe sampling without locks.

Iterators:

- Iterating `0..city_count` and shuffling in place monomorphizes into lean machine code. There’s no virtual dispatch in the hot path, and the optimizer can inline aggressively.

Memory control & locality:

- `Vec::collect()` builds a single, contiguous population buffer; each Individual holds its tour inline. This avoids scattered allocations and keeps initialization bandwidth-bound rather than synchronization-bound.



```rust
use rand::{SeedableRng, seq::SliceRandom};
use rand_chacha::ChaCha20Rng;
use rayon::prelude::*;

/// Parallel population initialization for TSP.
/// - Each Tour is a random permutation of 0..city_count
pub fn init_population_tsp(
    city_count: usize,
    population_size: usize,
    seed: u64,
) -> Vec<Individual<Tour>> {
    (0..population_size)
        .into_par_iter()
        .map(|i| {
            // Create a random seed
            let deriv = seed ^ (i as u64).wrapping_mul(0x9E37_79B9_7F4A_7C15);
            let mut rng = ChaCha20Rng::seed_from_u64(deriv);

            // Build a permutation in-place using Iterator::shuffle
            let mut order: Vec<usize> = (0..city_count).collect();
            order.shuffle(&mut rng);

            // Create the individual, set fitness to None
            Individual::new(Tour{ order }, None)
        })
        .collect()
}
```

2. Fitness Evaluation

- **Definition**: Compute objective value per individual (minimize tour length; minimize function value)..

- **Goal**: This is typically the bottleneck; optimizing here yields the biggest speedups.

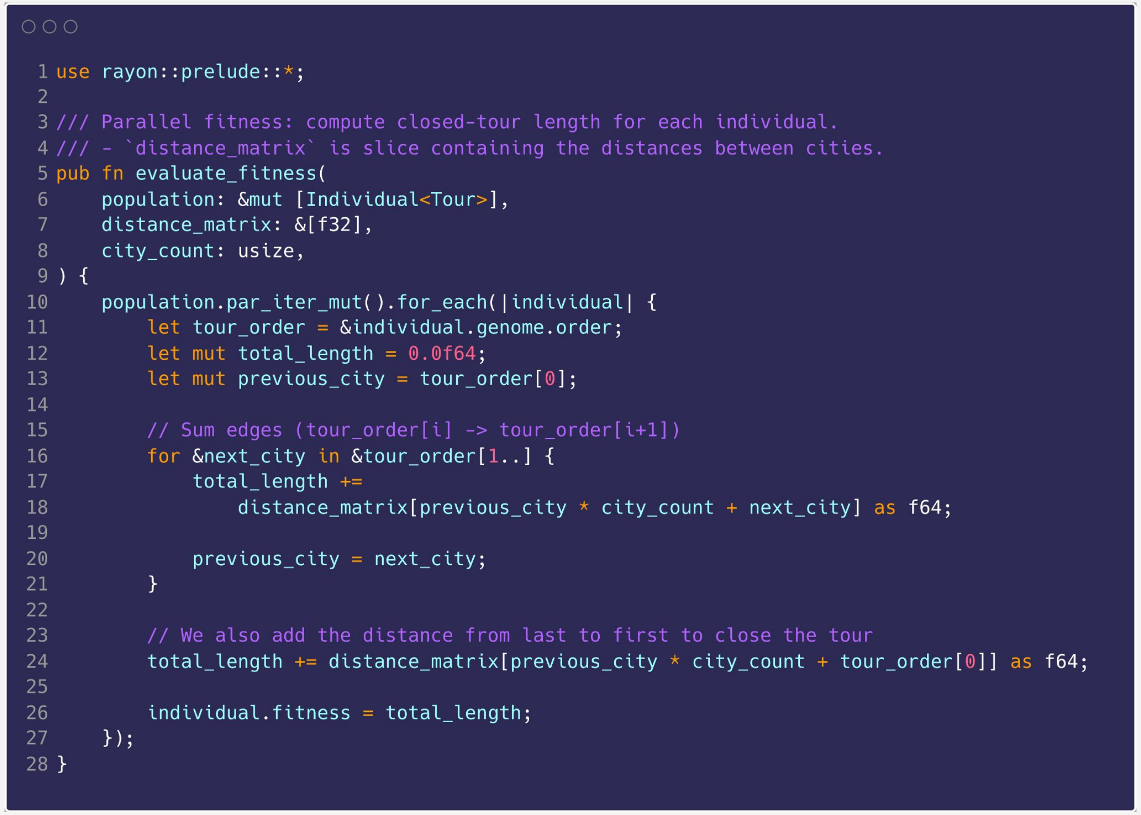
- **Examples:**
  - **TSP:** Random city permutations
    - One can use some exploitative greedy heuristics e.g. nearest-neighbor to seed quality.
  - **Continuous:** Uniform random within bounds.
- **Population size:** Tune for diversity vs. runtime. This can range from small (64) to large (2000+) depending on constraints

2. Fitness Evaluation - Rust

- **Parallelism: par_iter_mut()** gives multicore speed without data races.

- **Performance:**
  - Vec<f32> for the distance matrix + Vec<usize> tours → simple pointer math and good locality. This minimizes branch mispredictions and memory traffic in the critical path

  - **Benchmark-Ready:** The pure-compute nature of the function lends itself to consistent profiling + improvements with simd if necessary



```rust
use rayon::prelude::*;

/// Parallel fitness: compute closed-tour length for each individual.
/// - `distance_matrix` is slice containing the distances between cities.
pub fn evaluate_fitness(
    population: &mut [Individual<Tour>],
    distance_matrix: &[f32],
    city_count: usize,
) {
    population.par_iter_mut().for_each(|individual| {
        let tour_order = &individual.genome.order;
        let mut total_length = 0.0f64;
        let mut previous_city = tour_order[0];

        // Sum edges (tour_order[i] -> tour_order[i+1])
        for &next_city in &tour_order[1..] {
            total_length +=
                distance_matrix[previous_city * city_count + next_city] as f64;

            previous_city = next_city;
        }

        // We also add the distance from last to first to close the tour
        total_length += distance_matrix[previous_city * city_count + tour_order[0]] as f64;

        individual.fitness = total_length;
    });
}
```

3. Selection

- **Definition**: Choose parents from the current population according to fitness.

- **Goal**: Controls selection pressure:
  - too high → premature convergence
    too low → stagnation.

- **Examples:**
  - **TSP:** Random city permutations
    - One can use some exploitative greedy heuristics e.g. nearest-neighbor to seed quality.
  - **Continuous:** Uniform random within bounds.

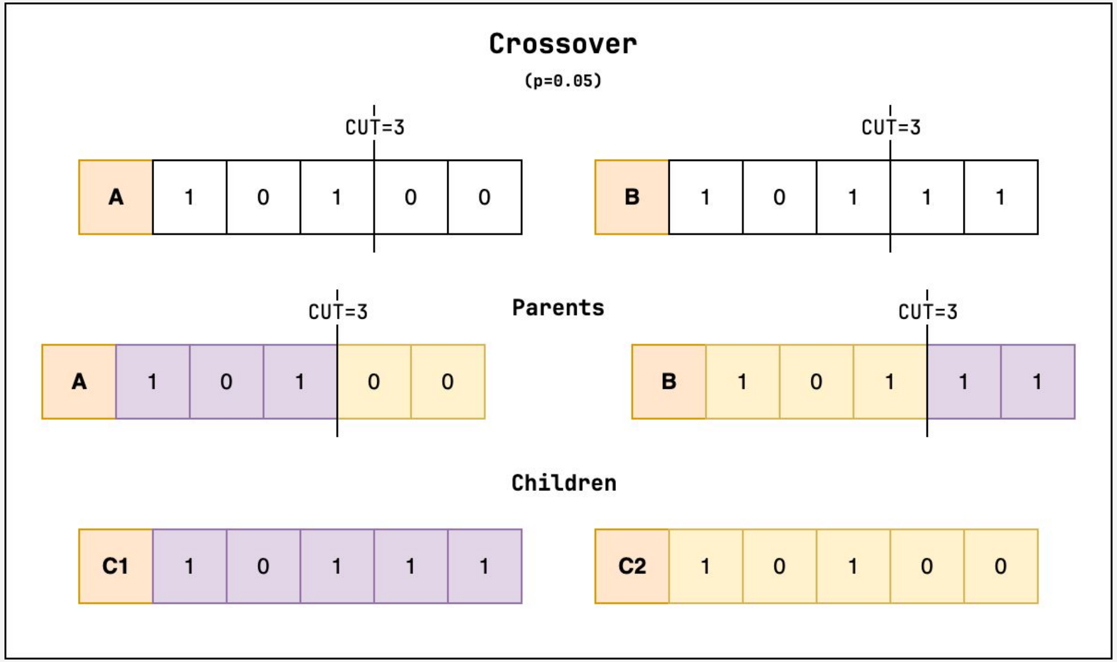
4. Variation: Crossover / Recombination

- **Definition**: Combine two parents to produce offspring.

- **Goal**: Transfers genetic building blocks to the next generation. All recombination must respect representation.

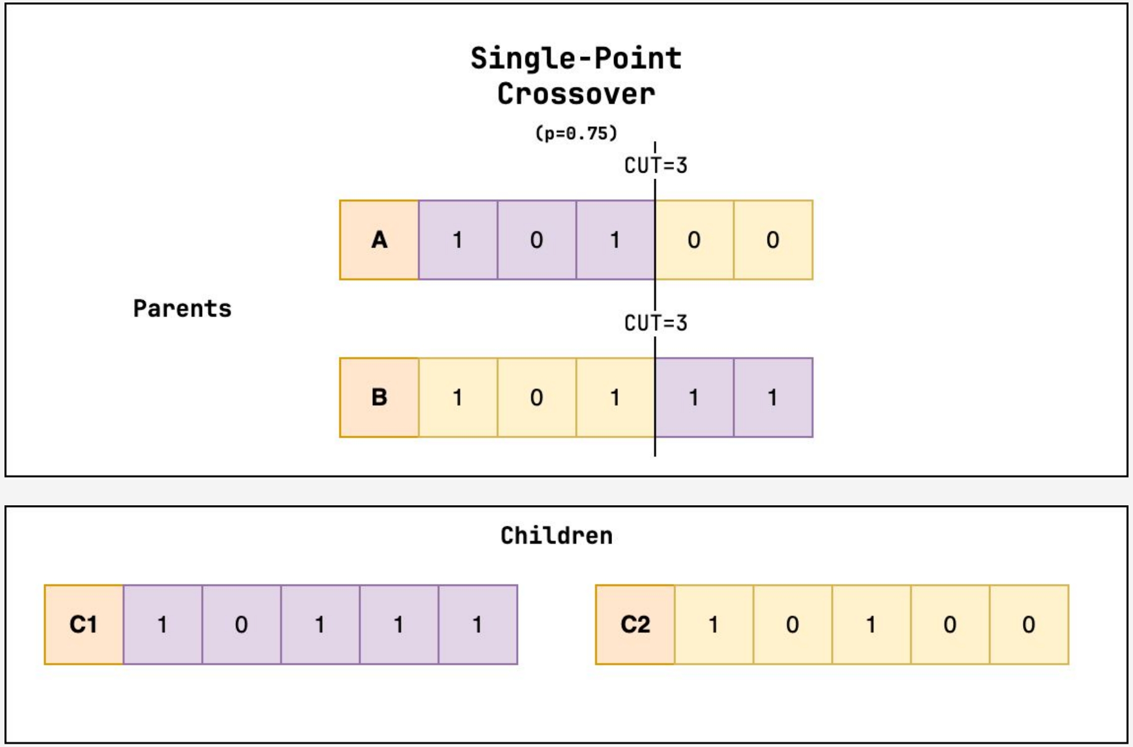
- **Examples:**
  - **TSP:** permutation-safe, no duplicates etc.
  - **Continuous:** Blending for floats.

- **Population size:** Tune for diversity vs. runtime. This can range from small (64) to large (2000+) depending on constraints



4a. Crossover: Single-Point

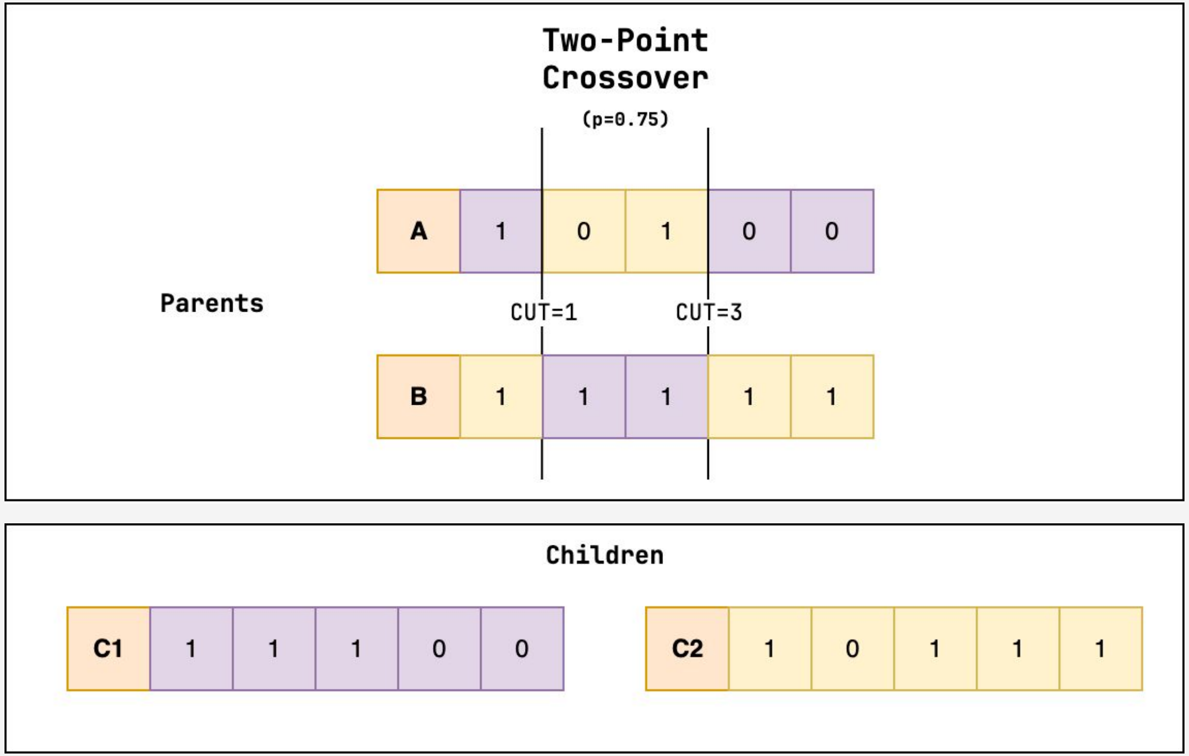
- **Algorithm**:
  - Select one random cut point along the chromosome.
  - Swap all genes after that point between two parents.
- **Best for**: Binary strings or fixed-length sequences where order isn't crucial.
- **Benefit**: Preserves large contiguous gene segments — good for "building blocks."



4b. Crossover: Two-Point

- **Algorithm**:
  - Pick two cut points and exchange the middle segment.

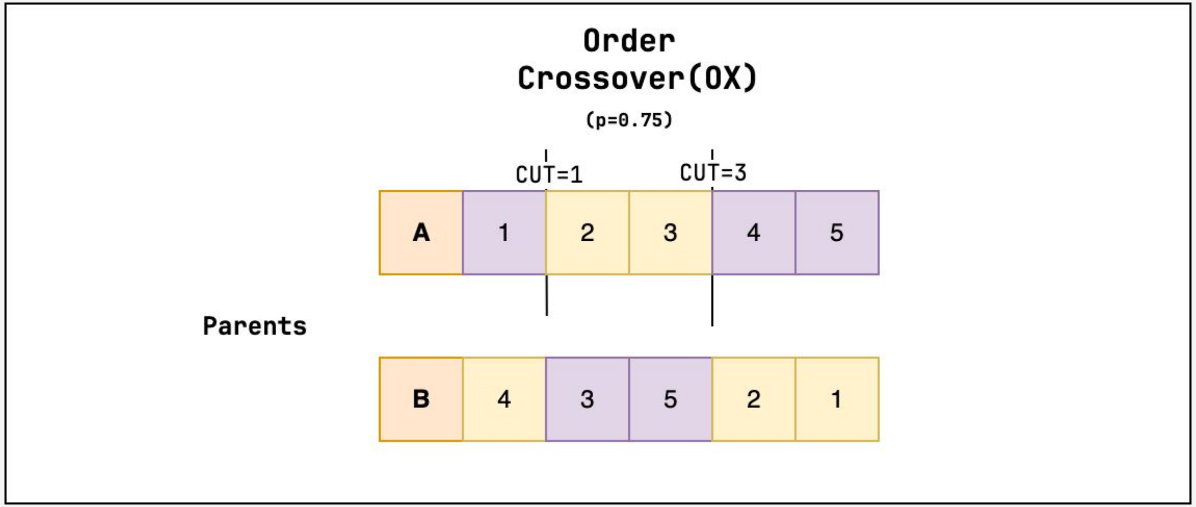
- **Best for**: Structured chromosomes where multiple features interact.

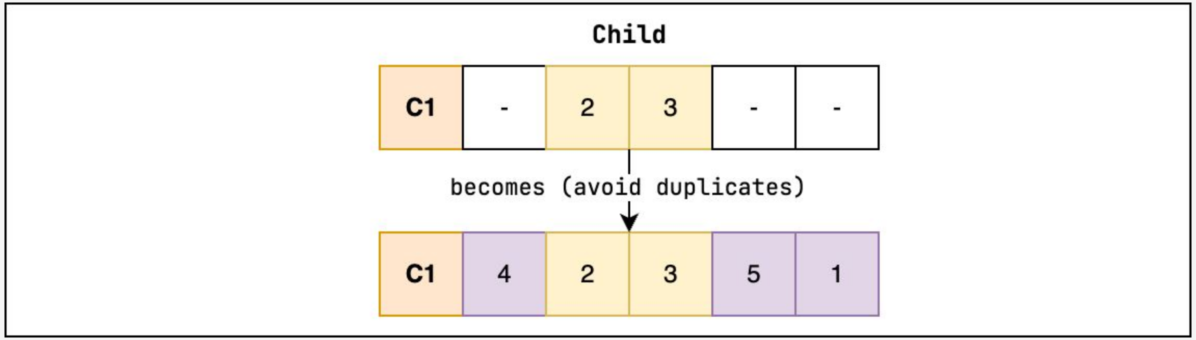


4c. Crossover: Order Crossover (OX)

- **Algorithm**:
  - Designed for permutation problems (e.g., TSP).
  - Preserve a subsequence from Parent1, then fill remaining cities from Parent2 in order.

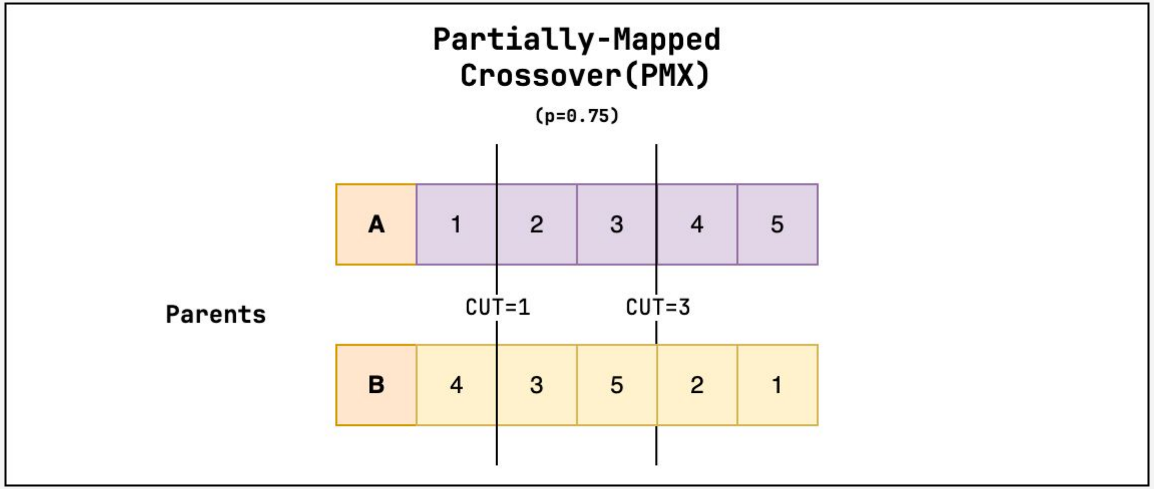
- **Benefit**: Order and relative position of elements.

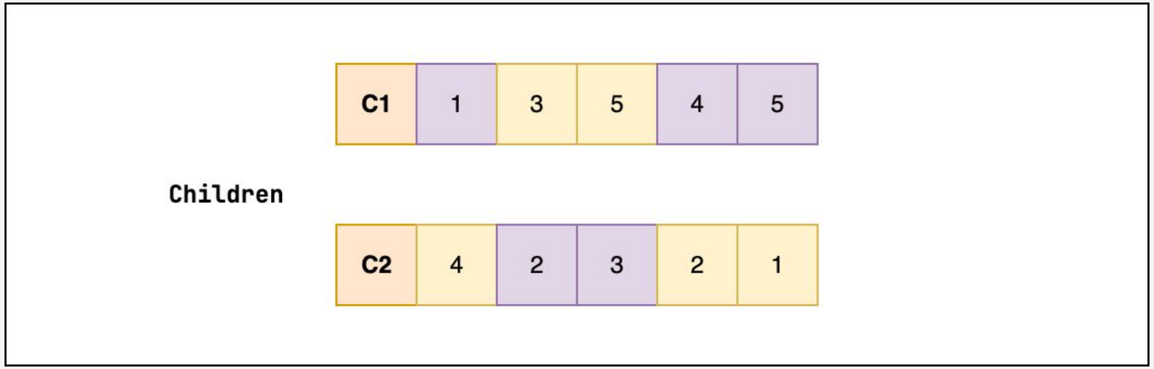




4d. Crossover: Partially-Mapped Crossover (PMX)

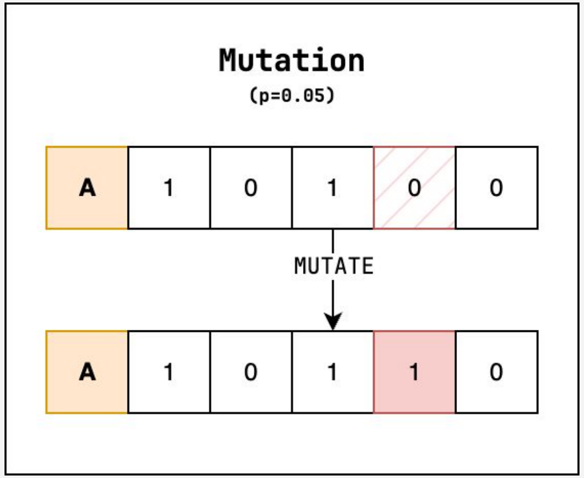
- **Algorithm**:
  - Another permutation crossover ensuring a mapping between swapped segments.
  - One of the most widely used operators for adjacency-type problems.
  - Proposed by Goldberg and Lingle as a recombination operator for the TSP

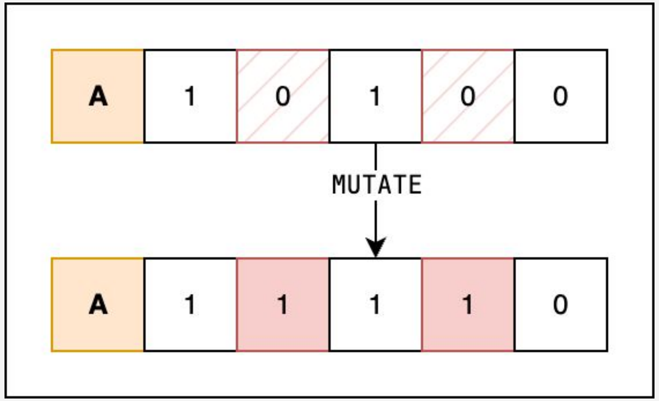




5. Mutation

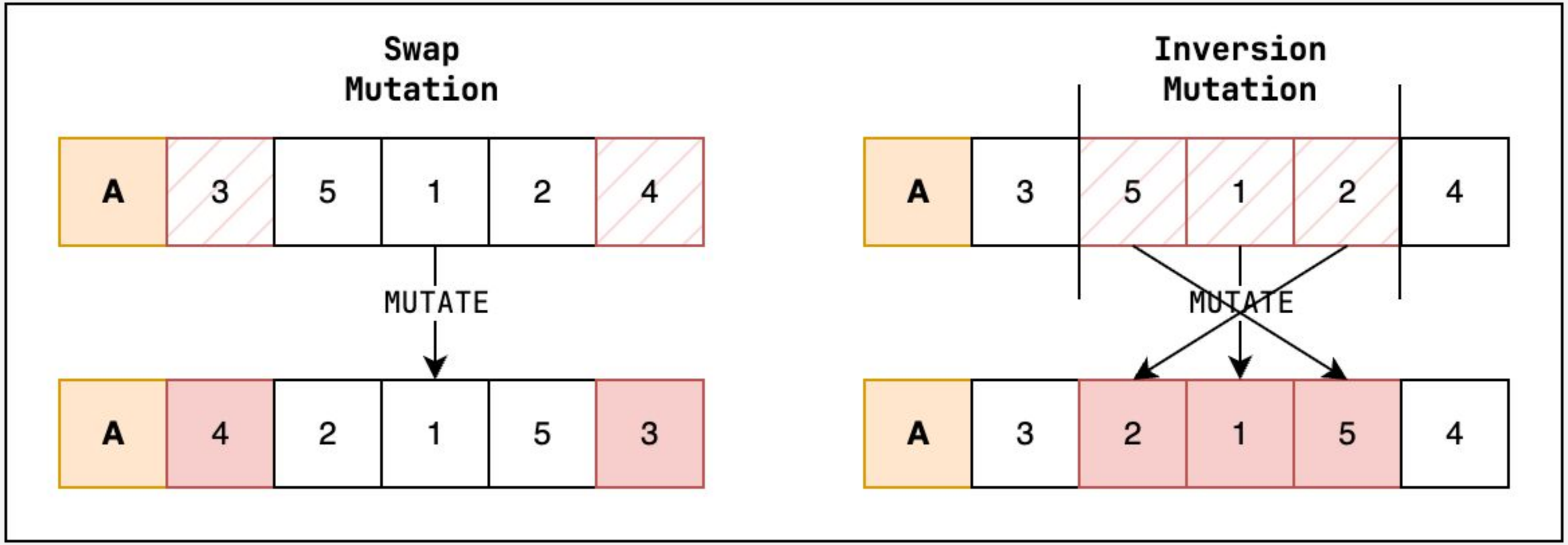
- **Definition**: Randomly perturb an individual to maintain exploration.
- **Goal**: Prevents genetic drift to a single basin; helps escape local optima.
- **Examples:**
  - **TSP:** Perform single-city swaps ensuring no duplicates
  - **Continuous:** Gaussian or polynomial per gene; consider σ decay.
- **Population size:** Tune for diversity vs. runtime. This can range from small (64) to large (2000+) depending on constraints





5. Mutation: TSP Mutation

- **Swap Mutation**: Perform single-city swaps ensuring no duplicates.
- **Inversion Mutation**: Pick two cut points, and reverse the cities.



6. Constraint Handling & Repair

- **Definition**: Ensure genomes remain valid and feasible after variation.

- **Goal**: Invalid genomes waste evaluations, crash operators, or bias selection.

- **Examples:**
    - **TSP:** Repair duplicates if any
    - **Continuous:** Branchless clamp to `[lo, hi]`; penalties or rejection for hard constraints..

7. Survivor Selection

- **Definition**: Replacement / Survivor Selection

- **Goal**: Balances preservation of good solutions (elitism) with continued exploration..

- **Examples:**
  - **TSP:** Repair duplicates if any
  - **Continuous:** Branchless clamp to [lo, hi]; penalties or rejection for hard constraints..

9. Diversity Maintenance

- **Definition**: Mechanisms to avoid population collapse to a single niche.

- **Goal**: Maintains coverage on multimodal landscapes (e.g., Rastrigin), improving robustness and peak discovery.

- **Examples:**
  - **TSP:** Repair duplicates if any
  - **Continuous:** Branchless clamp to `[lo, hi]`; penalties or rejection for hard constraints..

10. Termination Criteria

- **Definition**: Rules for stopping the run (budget, plateau, target achieved).
- **Goal**: Prevents wasted compute; enables fair comparisons between settings.
- **Examples:**
    - **TSP:** Repair duplicates if any
    - **Continuous:** Branchless clamp to `[lo, hi]`; penalties or rejection for hard constraints..

1. Traveling Salesman
2. Optimization Functions

Demos

Traveling Salesman Demo

Optimization Function Demo

Try it Yourself

https://rednuht.org/genetic_walkers/